ADOBE® INCOPY® CS6

# ADOBE INCOPY CS6 SCRIPTING GUIDE: VBSCRIPT

Adobe

*Adobe® InCopy® CS6 Scripting Guide: VBScript*

| Document Update Status (for entire document; see each chapter for chapter-specific update status) | | |
|---|---|---|
| CS6 | Updated | Throughout document, changed CS5 to CS6 and version 7.0 to 8.0. |

# Contents

# 1 Introduction

| Chapter Update Status | |
|---|---|
| CS6 | Unchanged |

This document shows how to do the following:

- Work with the Adobe® InCopy® scripting environment.
- Use advanced scripting features.
- Work with text and type in an InCopy document, including finding and changing text.
- Create dialog boxes and other user-interface items.
- Customize and add menus and create menu actions.
- Respond to user-interface events.
- Work with XML, from creating XML elements and importing XML to adding XML elements to a layout.

## How to use the scripts in this document

For the most part, the scripts shown in this document are not complete scripts. They are only fragments of scripts, and are intended to show only the specific part of a script relevant to the point being discussed in the text. You can copy the script lines shown in this document and paste them into your script editor, but you should not expect them to run without further editing. Note, in addition, that scripts copied out of this document may contain line breaks and other characters (due to the document layout) that will prevent them from executing properly.

A zip archive of all of the scripts shown in this document is available at the InCopy scripting home page, at: http://www.adobe.com/products/InCopy/scripting/index.html. After you have downloaded and expanded the archive, move the folders corresponding to the scripting language(s) of your choice into the Scripts Panel folder inside the Scripts folder in your InCopy folder. At that point, you can run the scripts from the Scripts panel inside InCopy.

### About the structure of the scripts

The script examples are all written using a common template that includes the functions “main,” “mySetup,” “mySnippet,” and “myTeardown.” We did this to simplify automated testing and publication—there’s no reason for you to construct your scripts this way. Most of the time, the part of the script you’ll be interested in will be inside the “mySnippet” function.

# For more information

For more information on InCopy scripting, you also can visit the InCopy Scripting User to User forum, at http://www.adobeforums.com. In the forum, scripters can ask questions, post answers, and share their newest scripts. The forum contains hundreds of sample scripts.

# About InCopy

InCopy is a collaborative, text-editing application developed for integrated use with Adobe InDesign®. InCopy enables you to track changes, add editorial notes, and fit copy tightly into the space designed for it. InCopy uses the same text-composition engine as InDesign, so InCopy and InDesign fit copy within a layout with identical composition.

InCopy is for the editorial environment. It allows editorial workflow participants to collaborate on magazines, newspapers, and corporate publishing, enabling concurrent text and layout editing. Its users are editors, writers, proofreaders, copy editors, and copy processors.

InCopy shares many panels and palettes with InDesign but also provides its own user-interface items.

# Relationships between InCopy and InDesign files

Relationships between InDesign and InCopy files are important because of the division of labor in a publication workflow that occurs when much of the same material is opened and modified in both applications.

There are two common scenarios for exporting from InCopy:

- You can export an (IDML based) ICML file.
- You can export an (INX based) INCX file.

There are two common scenarios for exporting from InDesign that involve InCopy in some way:

- Stories exported from InDesign as InCopy files are XML files or streams; the InCopyExport and InCopyWorkflow plug-ins loaded into InDesign provide this function. Some practical implications of this approach for InCopy files are that they are much smaller, they are faster over the network, they do not contain any page geometry, and data within the XML file or stream is available outside InDesign/InCopy (for search engines, database tools, and so on).
- Groupings within an article (such as a headline, byline, copy, graphics, or captions) also can be exported. InDesign and Incopy support the creation of groupings with assignment files, which handle file management by adding an additional file that tracks the other files. In essence, an assignment is a set of files whose contents are assigned to one person for some work to be done (for example, copy edit, layout, and/or writing). Any stories in an assignment are exported as InCopy files. Geometry information and the relationship of the files are held in the assignment file. InDesign allows the user to export a given set of stories by exporting into an assignment. InCopy opens all stories that are in an assignment together (as one unit). For details, see Chapter 10, "Assignments."

## Stories

Each InCopy file represents one story. An InDesign document containing several stories can be modularized to the same number of InCopy documents, through export. Those exported InDesign stories

contain a link, which may be viewed in the Links panel (InDesign) or the Assignments palette as assignment files (InCopy).

InCopy does not maintain a link to the InDesign document it is associated with (if one exists). InDesign maintains any links with InCopy files as bidirectional links.

Stories can be structured in XML. This means XML data can be contained within XML data. This feature can be used to design a data structure in which the raw text of a story is contained within an outer structure that contains data specific to InCopy (like styles).

Within InCopy, content can be saved in an ICML/INCX format or, if there is structure in the story, the logical structure can be exported in XML.

An ICML or INCX file can contain both InCopy data and marked-up text. If the file is exported as XML data, the data specific to InCopy is stripped out, leaving the marked-up content minus the information about how it is to be styled.

## Page geometry

InCopy files do not contain page geometry. When geometry is needed, it must be obtained from the InDesign document. InCopy can open InDesign documents and extract design information and links to the exported stories where needed. When page geometry is desired from within InCopy, assignment files can be supplied with it.

## Metadata

The Adobe Extensible Metadata Platform (XMP) provides a practical method for creating, interchanging, and managing metadata. InCopy files support XMP.

Just as InDesign provides the File > File Info command to view XMP data, InCopy provides the File > Content File Info command. System integrators can retain this data or strip it out during export.

Metadata added to stories by third-party software developers is preserved when incorporated into InDesign documents. Added metadata can be viewed within InDesign (from the File Info dialog box, available from the Links panel menu), as well as viewed within InCopy. Further, third-party software developers can add functionality to InDesign to view that metadata in a custom user interface.

An extensibility point exists for service providers to add metadata content to InCopy files. For more information, see Chapter 11, “XML.”

## The document model

InDesign documents are the basis for all content in InDesign. InCopy also uses InDesign documents, but they are not the default document type.

In both InDesign and InCopy, the basic document always is a database; in InCopy, however, this document may be an incomplete document. In InDesign, the main document typically is an opened InDesign file, but it also can be an opened INX or IDML file, which typically appears to be an unsaved InDesign document.

InCopy has other permutations. There is the basic InDesign file, as well as a new document with an InCopy story (or plain or RTF text) imported into it. Also, there are IDML- and INX-based assignment files, which have some part of an InDesign file stored in an XML file. The InDesign/InCopy document model corresponds to the base required model plug-in set, versioned against changes over time. It is important

that all IDML/INX scripting work in both InDesign and InCopy, so documents can be moved with high fidelity between the applications.

### User-interface differences

InDesign and InCopy share most of their panels, but InCopy has a smaller set and several additional toolbars along the top, left, and bottom screen borders. Most InCopy panels also can be docked on these bars, providing a smaller but always-visible view of the panel.

InCopy also has a custom window layout with multiple views, in a main window with three tabs: Galley view, Story view, and Layout view. Layout view is the InDesign window view. Galley and story views are simply the story-editor view, with and without accurate line endings, respectively.

## Design and architecture

### Story/file relationship

ICML is an IDML-based representation of an InCopy story. It represents the future direction of InDesign/InCopy and is an especially good choice if you need to edit a file outside of InDesign.

### ICML format

Each InCopy file or stream is in XML. An advantage of this is that InCopy files can be parsed easily and opened by any text editor.

### INCX format

INCX is an INX-based representation of an InCopy story. This format is not as readable as ICML, but it is still available to support INCX-based workflows.

### Document operations

InCopy provides default implementations of document operations (file actions) like New, Save, Save As, Save A Copy, Open, Close, Revert, and Update Design. All these InCopy file actions are in one plug-in (InCopyFileActions) in source-code form. Software developers or system integrators are expected to replace this with their own implementations, to customize the interaction for their workflow system.

### Using XMP metadata

Users can enter and edit metadata by choosing File > Content File Info. This metadata is saved in the InCopy file. Software developers and system integrators can create and store their own metadata using the XMP SDK.

# 2 Getting Started

| Chapter Update Status | | |
|---|---|---|
| CS6 | Updated | Removed or changed specific references to CS5. . |

Scripting is the most powerful feature in Adobe® InCopy®. No other feature—no tool, panel, or dialog box you see in the program's user interface—can save you as much time, trouble, and money as scripting.

This document is for every InCopy user. It does not matter if you havenever created a script before; this manual shows you how to get started. If you wrote scripts before for other applications, this manual shows you how to apply your knowledge to InCopy scripting. It covers installing and running an InCopy script, and it describes what InCopy scripting can and cannot do. It also discusses the software you need to get started writing your own scripts.

Almost anything you can do with the InCopy user interface, you can do with a script. You can enter and format text, find and change text, add notes, and print or export the pages of the document. Any action that can change a document or its contents can be scripted. There are even a few things that you can do in scripting that you cannot do using the user interface.

Scripts can create menus, add menu items, create and display dialogs and panels, and respond to your user-interface selections. Scripts can read and write text files, parse XML data, and communicate with other applications. Scripts can do everything from very small tasks (like setting a tab stop at the location of the text cursor) to providing complete features. You can start with very simple scripts that do only one thing and move on to scripts that automate your entire publishing workflow.

Most of the things scripting cannot do—like setting up a workspace or defining a set of keyboard shortcuts—are related to the user interface. In addition, scripts cannot add new kinds of objects to an InCopy document or add new, fundamental capabilities to the program, like a new text-composition engine. For that type of extensibility, you must to turn to the InCopy Software Development Kit (SDK), which shows you how to write compiled plug-ins using C++.

This document talks about Adobe InDesign® as well as InCopy, because InCopy almost always is used in conjunction with InDesign documents. In addition, InDesign and InCopy scripting are very similar. For more on InDesign scripting, see *Adobe InDesign Scripting Tutorial* and *Adobe InDesign Scripting Guide*.

## Installing scripts

To install an InCopy script, just put the script file in the Scripts Panel folder in the Scripts folder in your InCopy application folder.

Alternately, put the script in the Scripts Panel folder in your user-preferences folder. You can find your user preferences folder at the following locations, where `<username>` is your user name and ~ (tilde) is your system volume:

Windows® XP: `C:\Documents and Settings\<username>\Application Data\Adobe\InCopy\Version 8.0\<locale>\Scripts`

Windows® Vista: `C:\Users\<username>\App Data\Roaming\Adobe\InCopy\Version 8.0\<locale>\Scripts`

Once the script is in the folder, it appears in the Scripts panel inside InCopy (choose Window > Scripts to display the panel).

You also can put in the Scripts Panel folder aliases/shortcuts to scripts or folders containing scripts, and they will appear in the Scripts panel.

# Running scripts

To run a script, display the Scripts panel (choose Window > Scripts), then double-click the script name in the Scripts panel. Many scripts display user-interface items (like dialogs or panels) and display alerts if necessary.

# Using the scripts panel

The Scripts panel can run compiled or uncompiled AppleScripts (files with the file extension `.spt`, `.as`, or `.applescript`), JavaScripts (files with the file extension `.js` or `.jsx`), VBScripts (files with the extension `.vbs`), or executable programs from the Scripts panel.

To edit a script shown in the Scripts panel, hold down Option (Mac OS) or Alt (Windows) key and double-click the script's name. This opens the script in the editor you defined for the script file type.

To open the folder containing a script shown in the Scripts panel, hold down the Command (Mac OS) or Ctrl-Shift (Windows) keys and double-click the script's name. Alternately, choose Reveal in Finder (Mac OS) or Reveal in Explorer (Windows) from the Scripts panel menu. The folder containing the script opens in the Finder (Mac OS) or Explorer (Windows).

Scripts run as a series of actions, which means you can undo the changes the script made to a document by choosing Undo from the Edit menu. This can help you troubleshoot a script, as you can step backward through each change.

To add a keyboard shortcut for a script, choose Edit > Keyboard Shortcuts, select an editable shortcut set from the Set menu, then choose Product Area > Scripts. A list of the scripts in your Scripts panel appears. Select a script and assign a keyboard shortcut as you would for any other InCopy feature.

# VBScript language details

Visual Basic tutorial scripts are written in VBScript. We chose VBScript because no added software is required to run or edit VBScripts; you can edit them with any text editor (like Notepad) and run them using the InCopy Scripts panel.

Other versions of Visual Basic include Visual Basic 5 Control Creation Edition (CCE), Visual Basic 6, Visual Basic .NET, and Visual Basic 2008 Express Edition. Versions of Visual Basic prior to Visual Basic .NET work well with InCopy scripting; Visual Basic .NET and newer versions work less well (because they lack the Variant data type, which is used extensively in InCopy scripting).

Many applications contain Visual Basic for Applications (VBA), like Microsoft® Word, Microsoft Excel, Microsoft Visio, and AutoCAD. Although you can use VBA to create InCopy scripts, InCopy does not include VBA.

To use VBScript or Visual Basic for InCopy scripting in Windows, you must install InCopy from a user account that has Administrator privileges. After you complete the installation, any user can run InCopy

scripts, and any user with Power User or Administrator privileges can add scripts to the InCopy Scripts panel.

# Using the scripts in this document

To use any script from this document, you can either open the tutorial script file (the filename is given before each script) or copy the code shown in this chapter.

The script files are stored in a zip archive, `InCopyCS6ScriptingGuideScripts.zip`. When you uncompress the archive, you can move the folder containing the scripts written in the scripting language you want to use (AppleScript, JavaScript, or VBScript) to your Scripts Panel folder. Working with the script files is much easier than entering the script yourself or copying and pasting from this document.

If you do not have access to the script archive, you can enter the scripting code shown in this chapter. To do this:

1. Copy the script from this Adobe PDF document and paste it into a text editor (such as Notepad) or a VBScript editor (such as VBSEdit).
2. Save the script as a plain-text file in the Scripts Panel folder (see "Installing scripts" on page 11), using the file extension .vbs.
3. Choose Windows > Scripts to display the Scripts panel.
4. Double-click the script name in the Scripts panel to run the script.

Entering scripts manually will work only for the scripts shown in this chapter. The scripts shown in the other chapters are script fragments, not complete scripts. To run these scripts, you must use the scripts from the script archive.

**NOTE:** If you are copying and pasting scripts from this document, be aware that line breaks caused by the layout of the document can cause errors in your script. As it can be very difficult to find such errors, we recommend hat you use the scripts in the zip archive.

# Your first InCopy script

Next, we create an InCopy script that creates a new document, adds a text frame, then enters text in the text frame. While this seems trivial, it demonstrates how to do the following:

- Establish communication with InCopy.
- Create a new document.
- Add text to a story.

Start a text editor (for example, Notepad) and enter the following script (or open the `HelloWorld.vbs` tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Add
Set myStory = myDocument.Stories.Item(1)
myStory.Contents = "Hello World!"
```

Save the script as a plain-text file with the file extension `.vbs` in the Scripts Panel folder (see "Installing scripts" on page 11). To run the script, double-click the script name in the Scripts panel.

## Walking through the script

Here is a step-by-step analysis of what the Hello World script does.

1. Establish communication with the InCopy application object:

```
Set myInCopy = CreateObject("InCopy.Application")
```

2. Create a new document and a reference to the document:

```
Set myDocument = myInCopy.Documents.Add
```

3. Get a reference to the first story in the document (a standalone document always contains a story):

```
Set myStory = myDocument.Stories.Item(1)
```

4. Add text to the story by setting the contents property to a string.

```
myStory.Contents = "Hello World!"
```

# Scripting terminology and the InCopy object model

Now that you created your first InCopy script, it is time to learn more about the terminology of scripting languages in general and InCopy scripting in particular.

## Scripting terminology

First, let's review a few common scripting terms and concepts.

### Comments

Comments give you a way to add descriptive text to a script. The scripting system ignores comments as the script executes; this prevents comments from producing errors when you run your script. Comments are useful when you want to document the operation of a script (for yourself or someone else). In this document, we use comments in the tutorial scripts.

To include a comment in VBScript, type `Rem` (for "remark") or `'` (one straight quote) to the left of the comment. To make an entire line a comment, type the comment marker at the beginning of a line. For example:

```
Rem this is a comment
' and so is this
```

### Values

The point size of a text character, the contents of a note, and the filename of a document are examples of *values* used in InCopy scripting. Values are the data your scripts use to do their work.

The *type* of a value defines what sort of data the value contains. For example, the value type of the contents of a word is a text string; the value type of the leading of a paragraph is a number. Usually, the values used in scripts are numbers or text. The following table explains the value types most commonly used in InCopy scripting:

| Value Type | What it is | Example |
|---|---|---|
| Boolean | Logical True or False. | `True` |
| Integer | Whole numbers (no decimal points). Integers can be positive or negative. | `14` |
| Double | A high-precision number that can contain a decimal point. | `13.9972` |
| String | A series of text characters. Strings appear inside (straight) quotation marks. | `"I am a string"` |
| Array | A list of values (the values can be any type). | `Array("0p0", "0p0", "16p4", "20p6")` |

### Converting values from one type to another

VBScript provides ways to convert variable values from one type to another. The most common converstions involved converting numbers to strings (so you can enter them in text or display them in dialogs) or converting strings to numbers (so you can use them to set a point size or page location).

```
Rem To convert from a number to a string:
myNumber = 2
myString = cstr(myNumber)
Rem To convert from a string to an integer:
myString = "2"
myNumber = cInt(myString)
Rem If your string contains a decimal value, use "cDbl" rather than "cInt":
myNumber = cDbl(myString)
```

## Variables

A *variable* is a container for a value. They are called “variables” because the values they contain might change. A variable might hold a number, a string of text, or a reference to an InCopy object. Variables have names, and you refer to a variable by its name. To put a value into a variable, you *assign* the data to the variable.

In all examples and tutorial scripts that come with InCopy, all variables start with `my`. This enables you to easily differentiate variables we created in a script from scripting-language terms.

### Assigning a value to a variable

Assigning values or strings to variables is fairly simple, as shown in these examples:

```
myNumber = 10
myString = "Hello, World!"
Set myTextFrame = myDocument.Pages.Item(1).TextFrames.Add
```

Try to use descriptive names for your variables, like `firstPage` or `corporateLogo`, rather than `x` or `c`. This makes your script easier to read. Longer names do not affect the execution speed of the script.

Variable names must be one word, but you can use internal capitalization (like `myFirstPage`) or underscore characters (`my_first_page`) to create more readable names. Variable names cannot begin with a number, and they cannot contain punctuation or quotation marks.

**Array variables**

An Array is a container for a series of values:

```
myArray = Array(1, 2, 3, 4)
Rem In Visual Basic.NET: myArray = New Double (1, 2, 3, 4)
```

To refer to an item in an array, refer to its index in the array.In VBScript, the first item in an array is item 0:

```
myFirstArrayItem = myArray(0)
```

**NOTE:** The Visual Basic `OptionBase` statement can be used to set the first item of an array to item 1. In the examples in this document, the first item in an array is item 0, not item 1, because that is the default. If you set `OptionBase` to 1, you must adjust all array references in the sample scripts accordingly.

Arrays can include other arrays, as shown in the following examples:

```
myArray = Array(Array(0,0), Array(72, 72))
Rem In Visual Basic.NET: myArray = New Array(New Double(0,0), NewDouble (0,0))
```

**Finding the value type of a variable**

Sometimes, your scripts must make decisions based on the value type of an object. If you are working on a script that operates on a text selection, for example, you might want that script to stop if nothing is selected.

```
Rem Given a variable of unknown type, "myMysteryVariable"...
myType = TypeName(myMysteryVariable)
Rem myType will be a string corresponding to the variable type (e.g., "Rectangle")
```

## Operators

Operators use variables or values to perform calculations (addition, subtraction, multiplication, and division) and return a value. For example:

```
MyWidth/2
```

returns a value equal to half of the content of the variable `myWidth`.

You also can use operators to perform comparisons (equal to (`=`), not equal to(`<>`), greater than (`>`), or less than (`<`)). For example:

```
MyWidth > myHeight
```

returns the value `true` (or 1) if `myWidth` is greater than `myHeight`; otherwise, `false` (0).

In VBScript, use the ampersand (`&`) to concatenate (or join) two strings. For example:

```
"Pride " & "and Prejudice"
```

returns the string:

```
"Pride and Prejudice"
```

### Conditional statements

"If the size of the selected text is 12 points, set the point size to 10 points." This is an example of a *conditional statement*. Conditional statements make decisions; they give your scripts a way to evaluate something (like the color of the selected text, number of pages in the document, or date), then act according to the result. Most conditional statements start with `if`.

### Control structures

If you could talk to InCopy, you might say, "Repeat the following procedure 20 times." In scripting terms, this is a *control structure*. Control structures provide repetitive processes, or *loops*. The idea of a loop is to repeat an action over and over again, with or without changes between instances (or *iterations*) of the loop, until a specific condition is met. Control structures usually start with the `for`.

### Functions

*Functions* are scripting modules to which you can refer from within your script. Typically, you send a value or series of values to a function and get back another value or values. There is nothing special about the code used in functions; they are simply conveniences to avoid having to type the same lines of code repeatedly in your script. Functions start with `function`.

## Understanding the InDesign and InCopy object model

When you think about InCopy and InDesign documents, you probably organize the programs and their components in your mind. You know that paragraphs are contained by text frames, which in turn appear on a page. A page is a part of a spread, and one or more spreads make up a document. Documents contain colors, styles, layers, and master spreads. As you think about the objects in the documents you create, you intuitively understand that there is an order to them.

InDesign and InCopy "think" about the contents of a document the same way you do. A document contains pages, which contain page items (text frames, rectangles, ellipses, and so on). Text frames contain characters, words, paragraphs, and anchored frames; graphics frames contain images, EPSs, or PDFs; groups contain other page items. The things we mention here are the *objects* that make up an InDesign publication, and they are what we work with when we write InDesign and InCopy scripts.

Objects in your publication are arranged in a specific order: paragraphs are inside a story, which is inside a document, which is inside the InCopy application object. When we speak of an *object model* or a *hierarchy*, we are talking about this structure. Understanding the object model is key to finding the object you want to work with. Your best guide to InCopy scripting is your knowledge of InCopy itself.

Objects have *properties* (attributes). For example, the properties of a text object include the font used to format the text, point size, and leading applied to the text.

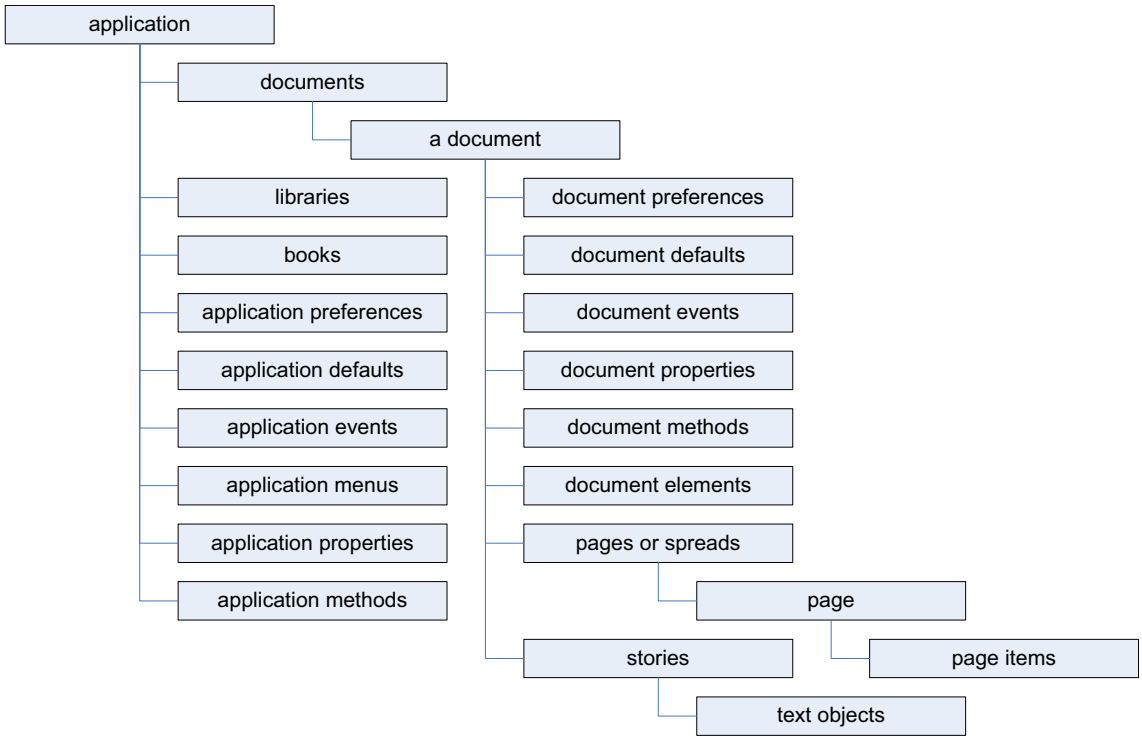
Properties have values; for example, the point size of text can be either a number (in points) or the string "Auto" for auto leading. The fill-color property of text can be set to a color, gradient, mixed ink, or swatch.

Properties also can be *read/write* or *read only*. Read/write properties can be set to other values; read-only properties cannot.

Objects also have *methods*. Methods are the verbs of the scripting world, the actions an object can perform. For example, the document object has print, export, and save methods.

Methods have *parameters*, or values that define the effect of the method. The open method, for example, has a parameter that defines the file you want to open.

The following block diagram is an overview of the InCopy object model. The diagram is not a comprehensive list of objects available to InCopy scripting; instead, it is a conceptual framework for understanding the relationships between the types of objects.




The objects in the diagram are explained in the following table:

| Term | What it represents: |
|---|---|
| Application | InCopy. |
| Application defaults | Application default settings, such as colors, paragraph styles, and object styles. Application defaults affect all new documents. |
| Application events | The things that happen as a user or script works with the application. Events are generated by opening, closing, or saving a document or choosing a menu item. Scripts can be triggered by events. |
| Application menus | The menus, submenus, and context menus displayed in the InCopy user interface. Scripts can be attached to menu choices and can execute menu actions. |
| Application methods | The actions the application can take; for example, finding and changing text, copying the selection, creating new documents, and opening libraries. |
| Application preferences | Examples are text preferences, PDF export preferences, and document preferences. Many preferences objects also exist at the document level. Just as in the user interface, application preferences are applied to new documents; document preferences change the settings of a specific document. |

| Term | What it represents: |
|---|---|
| Application properties | The properties of the application; for example, the full path to the application, the locale of the application, and the user name. |
| Books | A collection of open books. |
| Document | An InCopy document. |
| Document defaults | Document default settings, such as colors, paragraph styles, and text formatting defaults. |
| Document elements | For example, the stories, imported graphics, and pages of a document. The figure above shows pages and stories, because those objects are extremely important containers for other objects; however, document elements also include rectangles, ovals, groups, XML elements, and any other type of object you can import or create. |
| Document events | Events that occur at the document level, such as importing text. See *application events* in this table. |
| Document methods | The actions the document can take; for example, closing a document, printing a document, and exporting a document. |
| Document preferences | The preferences of a document, such as guide preferences, view preferences, or document preferences. |
| Document properties | For example, the document filename, number of pages, and zero-point location. |
| Documents | A collection of open documents. |
| Libraries | A collection of open libraries. |
| Page | One page in an InCopy document. |
| Page item | Any object you can create or place on a page. There are many types of page items, such as text frames, rectangles, graphic lines, and groups. |
| Pages or spreads | The pages or spreads in an InCopy document. |
| Stories | The text in an InCopy document. |
| Text objects | Characters, words, lines, paragraphs, and text columns are examples of text objects in an InCopy story. |

## Looking at the InCopy object model

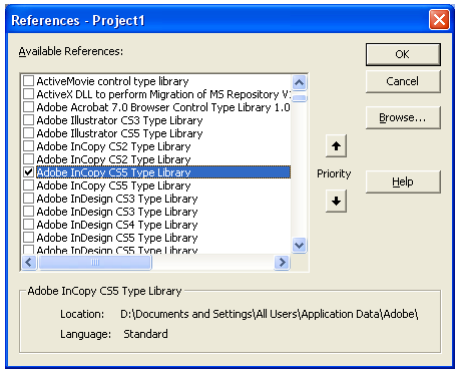
You can view the InCopy object model from inside your script-editing application. All reference information on objects and their properties and methods is stored in the model and can be viewed,

To view the InCopy object model, you need a VBScript editor/debugger, some version of Visual Basic, or an application that incorporates Visual Basic for Applications.

**Visual Basic 6**

To view the object model using Visual Basic 6:

1. Create a new Visual Basic project, then choose Project > References. Visual Basic displays the References dialog:

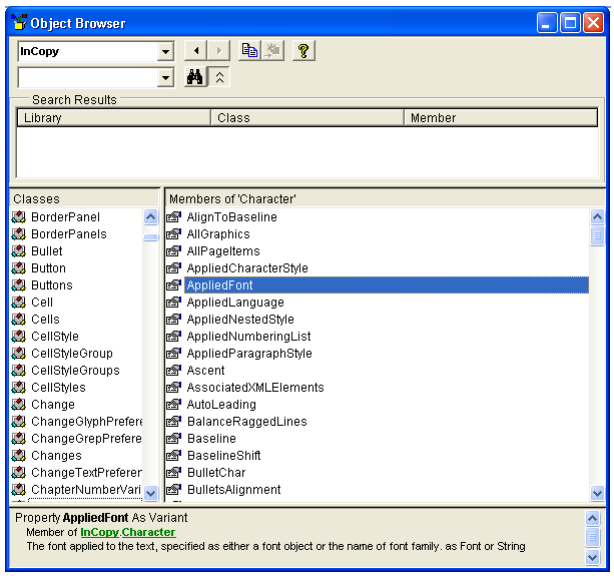



2. From the list of available references, select the Adobe InCopy CS6 Type Library option, and click OK. If the library does not appear in the list of available references, click Browse, then find and select the Resources for Visual Basic.tlb file. Usually this file is in `~\Documents and Settings\<username>\Application Data\Adobe\InCopy\Version 8.0\<locale>\Scripting Support\` (where *<username>* is your user name and ~ is your system volume). If necessary, search for the file. Once you find the file, click Open to add the reference to your project.

3. Choose View > Object Browser. Visual Basic displays the Object Browser dialog.

4. From the list of open libraries shown in the Project/Library menu, select InCopy. Visual Basic displays the objects that make up the InCopy object model.

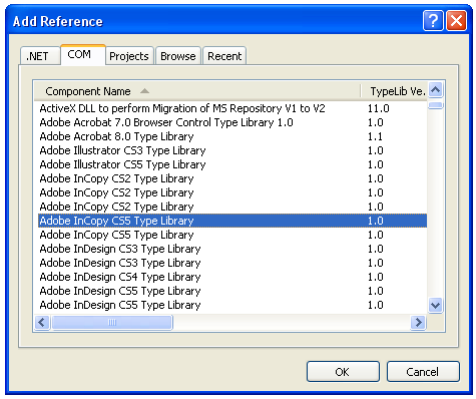
5. Click an object class. Visual Basic displays the object's properties and methods. For more information on a property or method, select the item; Visual Basic displays the definition of the item at the bottom of the Object Browser window:



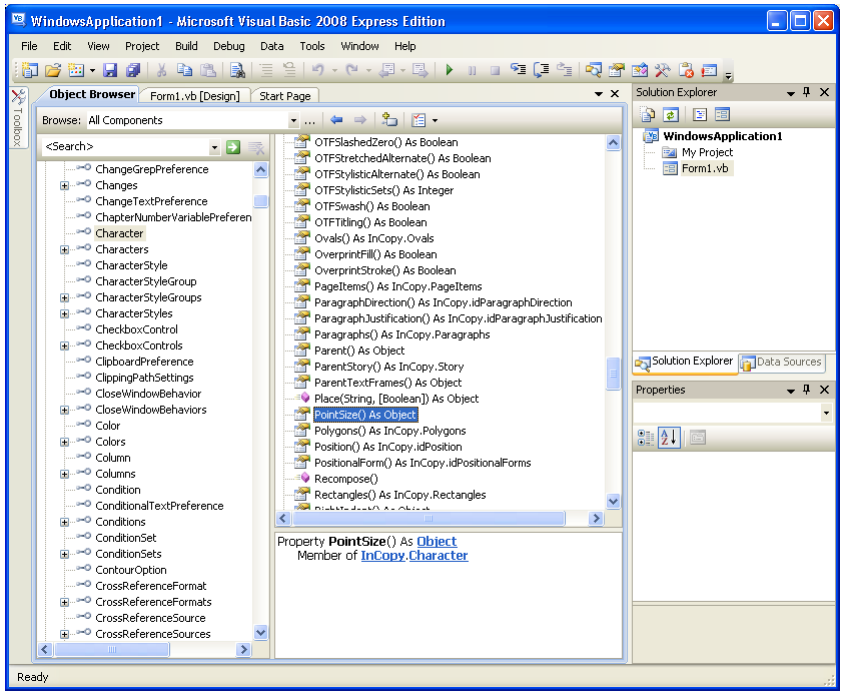

### Visual Basic Express 2008/Visual Basic.NET

To view the object model using Visual Basic Express 2008/Visual Basic.NET:

1. Create a new Visual Basic project, then choose Project > Add Reference. Visual Basic displays the Add Reference dialog.
2. Select the COM tab.
3. From the list of available references, select Adobe InCopy CS6 Type Library. Visual Basic adds the reference to the Selected Components list. If the library does not appear in the list of available references, click Browse, then find and select the Resources for Visual Basic.tlb file. Usually this file is in `~\Documents and Settings\<username>\Application Data\Adobe\InCopy\Version 8.0\<locale>\Scripting Support\` (where `<username>` is your user name). Once you find the file, click Open to add the reference to your project.



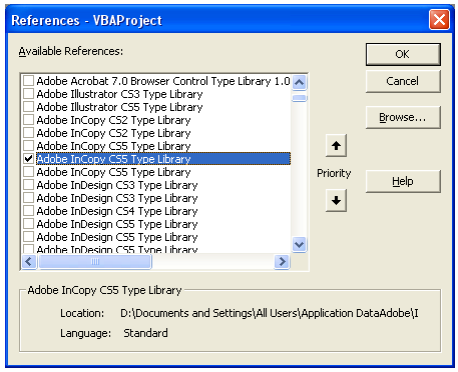

4. Click OK.
5. Choose View > Object Browser. Visual Basic displays the Object Browser tab.
6. From the list of open libraries shown in the Objects window, choose interop.incopy. Visual Basic displays the objects that make up the InCopy object model.
7. Click an object class. Visual Basic displays the object's properties and methods. For more information on a property or method, select the item; Visual Basic displays the definition of the item at the bottom of the Object Browser window:




### Visual Basic for Applications

To view the object model using Visual Basic for Applications from Microsoft Excel:

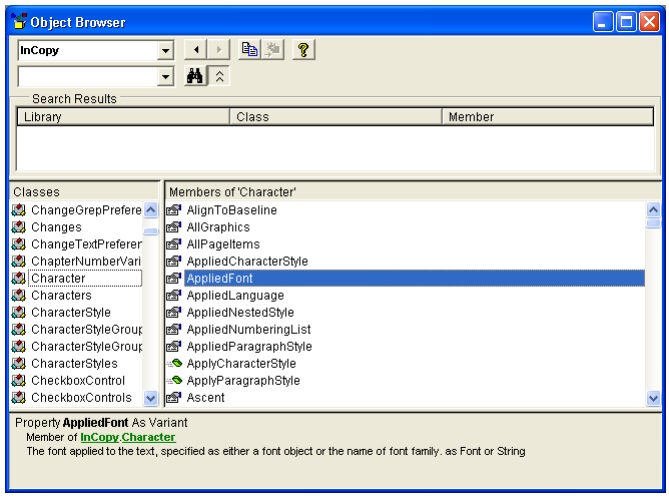
1. Start Excel.
2. Choose Tools > Macros > Visual Basic Editor. Excel displays the Visual Basic Editor window.
3. Choose Tools > References. The Visual Basic Editor displays the Add References dialog:




4. From the list of available references, select the Adobe InCopy CS6 Type Library, and click OK. Visual Basic adds the reference to the Selected Components list. If the library does not appear in the list of available references, click Browse, then find and select the Resources for Visual Basic.tlb file. Usually this file is in `~\Documents and Settings\<username>\Application Data\Adobe\InCopy\Version 8.0\<locale>\Scripting Support\` (where `<username>` is your user name). Once you find the file, click OK to add the reference to your project.
5. Choose View > Object Browser. The Visual Basic editor display the Object Browser window.

6. Choose InCopy from the Libraries pop-up menu. The Visual Basic editor displays a list of the objects in the InCopy object library.

7. Click an object name. The Visual Basic Editor displays the object's properties and methods of the object. For more information on a property or method, select the item; Visual Basic displays the definition of the item at the bottom of the Object Browser window:




# Measurements and positioning

All items and objects in InCopy are positioned on the page according to measurements you specify. It is useful to know how the InCopy coordinate system works and what measurement units it uses.

## Coordinates

InCopy, like every other page-layout and drawing program, uses simple, two-dimensional geometry to set the position of objects on a page or spread. The horizontal component of a coordinate pair is referred to as *x*; the vertical component, *y*. You can see these coordinates in the Transform panel or Control when you select an object using the Selection tool. As in the InCopy user interface, coordinates are measured relative to the current location of the ruler's zero point.

There is one difference between the coordinates used in InCopy and the coordinate system used in a Geometry textbook: on the InCopy vertical (or *y*) axis, coordinates *below* the zero point are positive numbers; coordinates *above* the zero point are negative numbers.

## Measurement units

When you send measurement values to InCopy, you can send numbers (for example, 14.65) or measurement strings (for example, "1p7.1"). If you send numbers, InCopy uses the publication's current units of measurement; if you send measurement strings (see the table below), InCopy uses the units of measurement specified in the string.

InCopy returns coordinates and other measurement values using the publication's current measurement units. In some cases, these units do not resemble the measurement values shown in the InCopy Transform panel. For example, if the current measurement system is picas, InCopy returns fractional values as decimals, rather than using the picas-and-points notation used by the Transform panel. So, for example,

"1p6," is returned as "1.5." InCopy does this because your scripting system would have trouble trying to perform arithmetic operations using measurement strings. For instance, trying to add "0p3.5" to "13p4" produces a script error, while adding .2916 to 13.333 (the converted pica measurements) does not.

If your script depends on adding, subtracting, multiplying, or dividing specific measurement values, you might want to set the corresponding measurement units at the beginning of the script. At the end of the script, you can set the measurement units back to whatever they were before you ran the script. Alternately, you can use measurement overrides, like many of the sample scripts. A measurement override is a string containing a special character, as shown in the following table:

| Override | Meaning | Example |
|---|---|---|
| c | ciceros (add didots after the c, if necessary) | 1.4c |
| cm | centimeters | .635cm |
| i (or in) | inches | .25i |
| mm | millimeters | 6.35mm |
| p | picas (add points after the p, if necessary) | 1p6 |
| pt | points | 18pt |

# Adding features to "Hello World"

Next, we create a new script that makes changes to the "Hello World" publication we created with our first script. Our second script demonstrates how to do the following:

- Get the active document.
- Change the formatting of the text in the first story.
- Add a note.

Either open the ImprovedHelloWorld tutorial script or follow these steps to create the script:

1. Start any text editor (for example, Notepad).
2. Make sure you have the document you created earlier open. If you closed the document without saving it, simply run the `HelloWorld.vbs` script again to make a new document.
3. Enter the following code:

```
Set myInCopy = CreateObject("InCopy.Application")
Rem Disable normal error handling (trying to get a non-
Rem existent font can cause errors).
On Error Resume Next
Rem Enter the name of a font on your system, if necessary.
Set myFont = myInCopy.Fonts.Item("Arial")
If Error.Number <> 0 Then
    Error.Clear
End If
Rem Resume normal error handling.
On Error Goto 0
Rem Get the active document and assign the result to the variable "myDocument"
Set myDocument = myInCopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Rem Change the font, size, and paragraph alignment.
If TypeName(myFont) <> "Nothing" Then
    myStory.AppliedFont = myFont
End If
myStory.PointSize = 48
myStory.Justification = idJustification.idCenterAlign
Rem Enter the note at the last insertion point of the story.
Set myInsertionPoint = myStory.InsertionPoints.Item(-1)
Set myNote = myInsertionPoint.Notes.Add
myNote.Texts.Item(1).Contents = "This is a Note."
```

4. Save the text as a plain-text file with the file extension `.vbs` in the Scripts folder (see "Installing scripts" on page 11).

Double-click the script name in the Scripts panel to run the new script.

# 3 Scripting Features

| Chapter Update Status | | |
|---|---|---|
| CS6 | Edited | "Script versioning" on page 27 and its three subsections have been updated, corrected, and clarified. |

This chapter covers scripting techniques that relate to InCopy's scripting environment. Almost every other object in the InCopy scripting model controls a feature that can change a document or the application defaults. By contrast, the features in this chapter control how scripts operate.

This document discusses the following:

- The `ScriptPreferences` object and its properties.
- Getting a reference to the executing script.
- Running scripts in prior versions of the scripting object model.
- Using the `DoScript` method to run scripts.
- Running scripts at InCopy start-up.

We assume that you have already read Chapter 2, "Getting Started" and know how to write, install, and run InCopy scripts in the scripting language of your choice.

## Script preferences

The `ScriptPreferences` object provides objects and properties related to the way InCopy runs scripts. The following table provides more detail on each property of the ScriptPreferences object:

| Property | Description |
|---|---|
| `EnableRedraw` | Turns screen redraw on or off while a script is running from the Scripts panel. |
| `ScriptsFolder` | The path to the scripts folder. |
| `ScriptsList` | A list of the available scripts. This property is an array of arrays, in the following form:<br>`[[fileName, filePath], ...]`<br>Where *`fileName`* is the name of the script file and *`filePath`* is the full path to the script. You can use this feature to check for the existence of a script in the installed set of scripts. |

| Property | Description |
|---|---|
| `UserInteractionLevel` | This property controls the alerts and dialogs that InCopy presents to the user. When you set this property to `idUserInteractionLevels.idNeverInteract`, InCopy does not display any alerts or dialogs; set it to `idUserInteractionLevels.idInteractWithAlerts` to enable alerts but disable dialogs; and set it to `idUserInteractionLevels.idInteractWithAll` to restore the normal display of alerts and dialogs. The ability to turn off alert displays is very useful when you are opening documents via script; often, InCopy displays an alert for missing fonts or linked graphics files. To avoid this alert, set the user-interaction level to `idUserInteractionLevels.idNeverInteract` before opening the document, then restore user interaction (set the property to `idUserInteractionLevels.idInteractWithAll`) before completing script execution. |
| `Version` | The version of the scripting environment in use. For more information, see "Script versioning" on page 27. Note that this property is *not* the same as the version of the application. |

# Getting the current script

You can get a reference to the current script using the `ActiveScript` property of the application object. You can use this property to help you locate files and folders relative to the script, as shown in the following example (from the ActiveScript tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
myActiveScript = myInCopy.ActiveScript
MsgBox ("The current script is: " & myActiveScript)
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
myParentFolder = myFileSystemObject.GetFile(myActiveScript).ParentFolder
MsgBox ("The folder containing the active script is: " & myParentFolder)
```

When you debug scripts using a script editor, the `ActiveScript` property returns an error. Only scripts run from the Scripts palette appear in the `ActiveScript` property.

# Script versioning

InCopy can run scripts using earlier versions of the InCopy scripting object model. To run an older script in a newer version of InCopy, you must consider the following:

- **Targeting —** Scripts must be targeted to the InCopy version in which they are being run (that is, the current version). The mechanics of targeting are language specific as described in "Targeting" on page 28.
- **Compilation —** This involves mapping the names in the script to the underlying script IDs, which are what InCopy understands. The mechanics of compilation are language specific as described in "Compilation" on page 28.
- **Interpretation —** This involves matching the IDs to the appropriate request handler within InCopy so that InCopy correctly interprets a script written for an earlier version of the scripting object model. To do this, either explicitly set the application's script preferences to the old object model within the

script (as shown in "Interpretation" on page 28) or run the script from a folder in the Scripts panel folder as follows:

| Folder | For InCopy version of scripts |
|---|---|
| `Version 8.0 Scripts` | CS6 |
| `Version 7.0 Scripts` | CS5 and CS5.5 |
| `Version 6.0 Scripts` | CS4 |
| `Version 5.0 Scripts` | CS3 |
| `Version 2.0 Scripts` | CS2 |

## Targeting

A script must always target the version of InCopy under which it is running (the current version), either explicitly or implicitly. Targeting is implicit when the script is launched from the Scripts panel.

Otherwise, explicit targeting for Visual Basic applications and VBScripts is done using the `CreateObject` method:

```
Rem Target InCopy CS6 Roman:
Set myApp = CreateObject("InCopy.Application.CS6")
Rem Target InCopy CS6 J:
Set myApp = CreateObject("InCopy.Application.CS6_J")
Rem Target the last version of InCopy that was launched:
Set myApp = CreateObject("InCopy.Application")
```

## Compilation

Compilation of Visual Basic applications may be versioned by referencing an earlier version of the type library. To generate an earlier version of the type library, use the PublishTerminology method, which is exposed on the Application object. The type library is generated into a folder (named with the version of the DOM) that is in the `Scripting Support` folder in your application's preferences folder. For example, to generate the CS5 dictionary into the `C:\Documents and Settings\<username>\Application Data\Adobe\InCopy\Version 8.0\<locale>\Scripting Support\7.0` folder:

```
Set myApp = CreateObject("InCopy.Application")
Rem Publish the InCopy CS5 type library (version 7.0 DOM)
myApp.PublishTerminology(7.0)
```

VBScripts are not precompiled. For compilation, InCopy generates and references the appropriate type library automatically, based on the version of the DOM that is set for interpretation.

## Interpretation

The InCopy application object contains a `ScriptPreferences` object, which allows a script to get or set the version of the scripting object model to use for interpreting scripts. The version defaults to the current version of the application and persists.

For example, to change the version of the scripting object model to CS5:

```
Set myInCopy = CreateObject("InCopy.Application")
Rem Set to 7.0 DOM
myInCopy.ScriptPreferences.Version = 7.0
```

# Using the DoScript method

The `DoScript` method gives a script a way to execute another script. The script can be a string of valid scripting code or a file on disk. The script can be in the same scripting language as the current script or another scripting language. The available languages vary by platform: on Mac OS, you can run either an AppleScript or a JavaScript; on Windows, you can run a VBScript or a JavaScript.

The `DoScript` method has many possible uses:

- Running a script in another language that provides a feature missing in your main scripting language. For example, VBScript lacks the ability to display a file or folder browser, which JavaScript has. AppleScript can be very slow to compute trigonometric functions (sine and cosine), but JavaScript performs these calculations rapidly. JavaScript does not have a way to query Microsoft® Excel for the contents of a specific spreadsheet cell, but both AppleScript and VBScript have this capability. In all these examples, the `DoScript` method can execute a snippet of scripting code in another language, to overcome a limitation of the language used for the body of the script.
- Creating a script "on the fly." Your script can create a script (as a string) during its execution, which it can then execute using the `DoScript` method. This is a great way to create a custom dialog or panel based on the contents of the selection or the attributes of objects the script creates.
- Embedding scripts in objects. Scripts can use the `DoScript` method to run scripts that were saved as strings in the `label` property of objects. Using this technique, an object can contain a script that controls its layout properties or updates its content according to certain parameters. Scripts also can be embedded in XML elements as an attribute of the element or as the contents of an element. See "Running scripts at start-up" on page 30.

## Sending parameters to DoScript

To send a parameter to a script executed by `DoScript`, use the following form (from the DoScriptParameters tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
myJavaScript = "alert(""First argument: "" + arguments[0] + ""\rSecond argument: "" +
arguments[1]);"
myInCopy.DoScript myJavaScript, idScriptLanguage.idJavascript, Array("Hello from
DoScript", "Your message here.")
myVBScript = "msgbox arguments(1), vbOKOnly, ""First argument: "" & arguments(0)"
myInCopy.DoScript myVBScript, idScriptLanguage.idVisualBasic, Array("Hello from
DoScript", "Your message here.")
```

## Returning values from DoScript

To return a value from a script executed by `DoScript`, you can use the `ScriptArgs` (short for "script arguments") object of the application. The following script fragment shows how to do this (for the complete script, see the DoScriptReturnValue tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
myJavaScript = "app.scriptArgs.setValue(""ScriptArgumentA"", ""This is the first
script argument value."");" & vbCr
myJavaScript = myJavaScript & "app.scriptArgs.setValue(""ScriptArgumentB"", ""This is
the second script argument value."");" & vbCr
myInCopy.DoScript myJavaScript, idScriptLanguage.idJavascript
myScriptArgumentA = myInCopy.ScriptArgs.GetValue("ScriptArgumentA")
myScriptArgumentB = myInCopy.ScriptArgs.GetValue("ScriptArgumentB")
MsgBox "ScriptArgumentA: " & myScriptArgumentA & vbCr & "ScriptArgumentB: " &
myScriptArgumentB
myVBScript = "Set myInCopy = CreateObject(""InCopy.Application"")" & vbCr
myVBScript = myVBScript & "myInCopy.ScriptArgs.SetValue ""ScriptArgumentA"", ""This is
the first script argument value."" & vbCr
myVBScript = myVBScript & "myInCopy.ScriptArgs.SetValue ""ScriptArgumentB"", ""This is
the second script argument value."""
myInCopy.DoScript myVBScript, idScriptLanguage.idVisualBasic
myScriptArgumentA = myInCopy.ScriptArgs.GetValue("ScriptArgumentA")
myScriptArgumentB = myInCopy.ScriptArgs.GetValue("ScriptArgumentB")
MsgBox "ScriptArgumentA: " & myScriptArgumentA & vbCr & "ScriptArgumentB: " &
myScriptArgumentB
```

# Running scripts at start-up

To run a script when InCopy starts, put the script in the Startup Scripts folder in the Scripts folder (for more information, see "Installing scripts" on page 11).

# 4 Text and Type

| Chapter Update Status | |
|---|---|
| CS6 | Unchanged |

Entering, editing, and formatting text make up the bulk of the time spent working on most InCopy documents. As a result, automating text and type operations can result in large productivity gains.

This tutorial shows how to script the most common operations involving text and type. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script. We also assume that you have some knowledge of working with text in InCopy and understand basic typesetting terms.

## Entering and importing text

This section covers the process of getting text into your InCopy documents. Just as you can type text into text frames and place text files using the InCopy user interface, you can create text frames, insert text into a story, or place text files using scripting.

### Stories and text frames

All text in an InCopy layout is part of a story, and every story can contain one or more text frames. If you are working with a standalone InCopy document, the document contains one story, and InCopy adds text frames only when necessary to display the text of the story. This also is true for stories exported from InDesign as InCopy stories (`.icml` files).

When you work with an InCopy story within an InDesign document, the document can contain any number of stories, and you will see the text frames as they were created in the InDesign layout. Unlike InDesign, InCopy cannot add new text frames using scripting.

For more on understanding the relationships between text objects in an InCopy document, see "Text objects" on page 40.

### Adding text to a story

To add text to a story, use the `contents` property. The following sample script uses this technique to add text at the end of a story (for the complete script, see AddText):

```
Set myDocument = myInCopy.Documents.Add
Rem Add text to the default story.
Set myStory = myDocument.Stories.Item(1)
myStory.Contents = "This is the first paragraph of example text."
Rem To add more text to the story, we'll use the last insertion point
Rem in the story. ("vbCr" is a return character in VBScript.)
Set myInsertionPoint = myStory.InsertionPoints.Item(-1)
myInsertionPoint.Contents = vbCr & "This is the second paragraph."
```

## Replacing text

The following script replaces a word with a phrase, by changing the contents of the appropriate object (for the complete script, see ReplaceWord):

```
Rem Enters text in the default story and then replaces
Rem a word in the story with a different phrase.
Set myDocument = myInCopy.Documents.Add
Rem Add text to the default story.
Set myStory = myDocument.Stories.Item(1)
myStory.Contents = "This is some example text."
Rem Replace the third word "some" with the phrase
Rem "a little bit of".
myStory.Words.Item(3).Contents = "a little bit of"
```

The following script replaces the text in a paragraph (for the complete script, see ReplaceText):

```
Rem Enters text in the default story, and then replaces
Rem the text in the second paragraph.
Set myDocument = myInCopy.Documents.Add
Set myStory = myDocument.Stories.Item(1)
myStory.Contents = "Paragraph 1." & vbCr & "Paragraph 2." & vbCr & "Paragraph 3." & vbCr
Rem Replace the text in the second paragraph without replacing
Rem the return character at the end of the paragraph. To do this,
Rem we'll use the ItemByRange method.
Set myStartCharacter = myStory.Paragraphs.Item(2).Characters.Item(1)
Set myEndCharacter = myStory.Paragraphs.Item(2).Characters.Item(-2)
Set myText = myStory.Texts.ItemByRange(myStartCharacter, myEndCharacter).Item(1)
myText.Contents = "This text replaces the text in paragraph 2."
```

In the preceding script, we used the `ItemByRange` method to get a reference to the text of the paragraph (excluding the return character at the end of the paragraph), as a single text object. We excluded the return character, because deleting the return might change the paragraph style applied to the paragraph. To use the `ItemByRange` method, we used the texts collection of the story but supplied two characters—the starting and ending characters of the paragraph—as parameters. If we had used `myTextFrame.ParentStory.Characters.ItemByRange`, InCopy would have returned a collection of `Character` objects. We wanted one `Text` object, so we could replace the contents in one action.

## Inserting special characters

Because most VBScript editors support Unicode, you can simply enter Unicode characters in text strings you send to InCopy. Alternately, you can use an InCopy shortcut to explicitly enter Unicode characters by their glyph ID number: <*nnnn*> (where *nnnn* is the Unicode code for the character). The following script shows several ways to enter special characters (for the complete script, see SpecialCharacters):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Add
Set myStory = myDocument.Stories.Item(1)
Rem Entering special characters directly.
myStory.Contents = "Registered trademark: ®" & vbCr & "Copyright: ©" & vbCr &
"Trademark: ™" & vbCr
Rem Entering special characters by their Unicode glyph ID value:
myStory.InsertionPoints.Item(-1).Contents = "Not equal to: <2260>" & vbCr
myStory.InsertionPoints.Item(-1).Contents = "Square root: <221A>" & vbCr
myStory.InsertionPoints.Item(-1).Contents = "Section symbol: <00B6>" & vbCr
Rem Entering InCopy special characters by their enumerations:
myStory.InsertionPoints.Item(-1).Contents = "Automatic page number marker:"
myStory.InsertionPoints.Item(-1).Contents = idSpecialCharacters.idAutoPageNumber
myStory.InsertionPoints.Item(-1).Contents = vbCr
myStory.InsertionPoints.Item(-1).Contents = "Section symbol:"
myStory.InsertionPoints.Item(-1).Contents = idSpecialCharacters.idSectionSymbol
myStory.InsertionPoints.Item(-1).Contents = vbCr
myStory.InsertionPoints.Item(-1).Contents = "En dash:"
myStory.InsertionPoints.Item(-1).Contents = idSpecialCharacters.idEnDash
myStory.InsertionPoints.Item(-1).Contents = vbCr
```

The easiest way to find the Unicode ID for a character is to use the Glyphs palette in InCopy (choose Type > Glyphs to display the palette)—move the cursor over a character in the palette, and InCopy will display its Unicode value. You can find out more about Unicode by visiting http://www.unicode.org.

# Placing text and setting text-import preferences

In addition to entering text strings, you can place text files created with word processors and text editors. The following script shows you how to place a text file in the default story of a new document (for the complete script, see PlaceTextFile):

```
Rem Places a text file in the default story of a new document.
Set myInCopy = CreateObject("InCopy.Application")
Rem Create an example document.
Set myDocument = myInCopy.Documents.Add
Rem Parameters for Page.place():
Rem File as File object,
Rem [ShowingOptions as Boolean = False]
Rem You'll have to fill in a valid file path on your system.
myDocument.Stories.Item(1).InsertionPoints.Item(1).Place "c:\test.txt", False
```

To specify the import options for the specific type of text file you are placing, use the corresponding import-preferences object. The following script shows how to set text-import preferences (for the complete script, see TextImportPreferences). Comments in the script show the possible values for each property.

```
Rem TextImportPreferences.vbs
Rem An InCopy CS6 VBScript
Rem
Set myInCopy = CreateObject("InCopy.Application")
Rem Sets the text import filter preferences.
With myInCopy.TextImportPreferences
    Rem Options for characterSet:
    Rem idTextImportCharacterSet.idAnsi
    Rem idTextImportCharacterSet.idChineseBig5
    Rem idTextImportCharacterSet.idGB18030
    Rem idTextImportCharacterSet.idGB2312
    Rem idTextImportCharacterSet.idKSC5601
    Rem idTextImportCharacterSet.idMacintoshCE
    Rem idTextImportCharacterSet.idMacintoshCyrillic
    Rem idTextImportCharacterSet.idMacintoshGreek
    Rem idTextImportCharacterSet.idMacintoshTurkish
    Rem idTextImportCharacterSet.idRecommendShiftJIS83pv
    Rem idTextImportCharacterSet.idShiftJIS90ms
    Rem idTextImportCharacterSet.idshiftJIS90pv
    Rem idTextImportCharacterSet.idUTF8
    Rem idTextImportCharacterSet.idUTF16
    Rem idTextImportCharacterSet.idWindowsBaltic
    Rem idTextImportCharacterSet.idWindowsCE
    Rem idTextImportCharacterSet.idWindowsCyrillic
    Rem idTextImportCharacterSet.idWindowsEE
    Rem icTextImportCharacterSet.idWindowsGreek
    Rem idTextImportCharacterSet.idWindowsTurkish
    .CharacterSet = idTextImportCharacterSet.idUTF16
    .ConvertSpacesIntoTabs = True
    .SpacesIntoTabsCount = 3
    Rem The dictionary property can take any of the following
    Rem language names (as strings):
    Rem Bulgarian
    Rem Catalan
    Rem Croatian
    Rem Czech
    Rem Danish
    Rem Dutch
    Rem English: Canadian
    Rem English: UK
    Rem English: USA
    Rem English: USA Legal
    Rem English: USA Medical
    Rem Estonian
    Rem Finnish
    Rem French
    Rem French: Canadian
    Rem German: Reformed
    Rem German: Swiss
    Rem German: Traditional
    Rem Greek
    Rem Hungarian
    Rem Italian
    Rem Latvian
    Rem Lithuanian
    Rem Neutral
    Rem Norwegian: Bokmal
    Rem Norwegian: Nynorsk
    Rem Polish
    Rem Portuguese
```

```
    Rem Portuguese: Brazilian
    Rem Romanian
    Rem Russian
    Rem Slovak
    Rem Slovenian
    Rem Spanish: Castilian
    Rem Swedish
    Rem Turkish
    .Dictionary = "English: USA"
    Rem platform options:
    Rem idImportPlatform.idMacintosh
    Rem idImportPlatform.idPC
    .Platform = idImportPlatform.idPC
    .StripReturnsBetweenLines = True
    .StripReturnsBetweenParagraphs = True
    .UseTypographersQuotes = True
End With
```

The following script shows how to set tagged text-import preferences (for the complete script, see TaggedTextImportPreferences):

```
Rem Sets the tagged text import filter preferences.
With myInCopy.TaggedTextImportPreferences
    RemoveTextFormatting = False
    Rem StyleConflict property can be:
    Rem idStyleConflict.idPublicationDefinition
    Rem idStyleConflict.idTagFileDefinition
    .StyleConflict = idStyleConflict.idPublicationDefinition
    .UseTypographersQuotes = True
End With
```

The following script shows how to set Word and RTF import preferences (for the complete script, see WordRTFImportPreferences):

```
With myInCopy.WordRTFImportPreferences
    Rem convertPageBreaks property can be:
    Rem idConvertPageBreaks.idColumnBreak
    Rem idConvertPageBreaks.idNone
    Rem idConvertPageBreaks.idPageBreak
    .ConvertPageBreaks = idConvertPageBreaks.idNone
    Rem convertTablesTo property can be:
    Rem idConvertTablesOptions.idUnformattedTabbedText
    Rem idConvertTablesOptions.idUnformattedTable
    .ConvertTablesTo = idConvertTablesOptions.idUnformattedTable
    .ImportEndnotes = True
    .ImportFootnotes = True
    .ImportIndex = True
    .ImportTOC = True
    .ImportUnusedStyles = False
    .PreserveGraphics = False
    .PreserveLocalOverrides = False
    .PreserveTrackChanges = False
    .RemoveFormatting = False
    Rem resolveCharacterStyleClash and resolveParagraphStyleClash properties can be:
    Rem idResolveStyleClash.idResolveClashAutoRename
    Rem idResolveStyleClash.idResolveClashUseExisting
    Rem idResolveStyleClash.idResolveClashUseNew
    .ResolveCharacterStyleClash =
    idResolveStyleClash.idResolveClashUseExisting
    .ResolveParagraphStyleClash =
    idResolveStyleClash.idResolveClashUseExisting
    .UseTypographersQuotes = True
End With
```

The following script shows how to set Excel import preferences (for the complete script, see ExcelImportPreferences):

```
With myInCopy.ExcelImportPreferences
    Rem alignmentStyle property can be:
    Rem AlignmentStyleOptions.centerAlign
    Rem AlignmentStyleOptions.leftAlign
    Rem AlignmentStyleOptions.rightAlign
    Rem AlignmentStyleOptions.spreadsheet
    .AlignmentStyle = idAlignmentStyleOptions.idSpreadsheet
    .DecimalPlaces = 4
    .PreserveGraphics = False
    Rem Enter the range you want to import as "start cell:end cell".
    .RangeName = "A1:B16"
    .SheetIndex = 1
    .SheetName = "pathpoints"
    .ShowHiddenCells = False
    Rem tableFormatting property can be:
    Rem idTableFormattingOptions.idExcelFormattedTable
    Rem idTableFormattingOptions.idExcelUnformattedTabbedText
    Rem idTableFormattingOptions.idExcelUnformattedTable
    .TableFormatting = idTableFormattingOptions.idExcelFormattedTable
    .UseTypographersQuotes = True
    .ViewName = ""
End With
```

# Exporting text and setting text-export preferences

The following script shows how to export text from an InCopy document. You must use text or story objects to export in text-file formats; you cannot export all the text in a document in one operation. (For the complete script, see ExportTextFile.)

```
Set myDocument = InCopy.Documents.Add
Set myStory = myDocument.Stories.Item(0)
Rem Fill the story with placeholder text.
Set myTextFrame = myStory.TextContainers.Item(1)
myTextFrame.Contents = idTextFrameContents.idPlaceholderText
Rem Text export method parameters:
Rem Format as idExportFormat
Rem To As File
Rem [ShowingOptions As Boolean = False]
Rem Format parameter can be:
Rem idExportFormat.idInCopyCSDocument
Rem idExportFormat.idInCopyCSDocument
Rem idExportFormat.idPDFType
Rem idExportFormat.idRTF
Rem idExportFormat.idTaggedText
Rem idExportFormat.idTextType
Rem
Rem Export the story as text. You must fill in a valid file path on your system.
myStory.Export idExportFormat.idTextType, "C:\test.txt"
```

The following example shows how to export a specific range of text. (We omitted the `myGetBounds` function from this listing; see the ExportTextRange tutorial script.)

```
Set myDocument = myInCopy.Documents.Add
Set myStory = myDocument.Stories.Item(1)
Rem Fill the story with placeholder text.
Set myTextFrame = myStory.TextContainers.Item(1)
myTextFrame.Contents = idTextFrameContents.idPlaceholderText
Set myStartCharacter = myStory.Paragraphs.Item(1).Characters.Item(1)
Set myEndCharacter = myStory.Paragraphs.Item(1).Characters.Item(-1)
Set myText = myStory.Texts.ItemByRange(myStartCharacter, myEndCharacter).Item(1)
Rem Format as idExportFormat
Rem To As File
Rem [ShowingOptions As Boolean = False]
Rem
Rem Format parameter can be:
Rem idExportFormat.idInCopyCSDocument
Rem idExportFormat.idInCopyDocument
Rem idExportFormat.idPDFType
Rem idExportFormat.idRTF
Rem idExportFormat.idTaggedText
Rem idExportFormat.idTextType
Rem
Rem Export the text range. You must fill in a valid file path on your system.
myText.Export idExportFormat.idTextType, "C:\test.txt"
```

To specify the export options for the specific type of text file you are exporting, use the corresponding export-preferences object. The following script sets text-export preferences (for the complete script, see TextExportPreferences):

```
Rem Sets the text export filter preferences.
With myInCopy.TextExportPreferences
    Rem Options for characterSet:
    Rem idTextExportCharacterSet.idUTF8
    Rem idTextExportCharacterSet.idUTF16
    Rem idTextExportCharacterSet.idDefaultPlatform
    .CharacterSet = idTextExportCharacterSet.idUTF16
    Rem platform options:
    Rem idImportPlatform.idMacintosh
    Rem idImportPlatform.idPC
    .Platform = idImportPlatform.idPC
End With
```

The following script sets tagged text-export preferences (for the complete script, see TaggedTextExportPreferences):

```
Rem Sets the tagged text export filter preferences.
With myInCopy.TaggedTextExportPreferences
    Rem Options for characterSet:
    Rem idTagTextExportCharacterSet.idAnsi
    Rem idTagTextExportCharacterSet.idASCII
    Rem idTagTextExportCharacterSet.idGB18030
    Rem idTagTextExportCharacterSet.idKSC5601
    Rem idTagTextExportCharacterSet.idShiftJIS
    Rem idTagTextExportCharacterSet.idUTF8
    Rem idTagTextExportCharacterSet.idUTF16
    .CharacterSet = idTagTextExportCharacterSet.idUTF16
    Rem tagForm options:
    Rem idTagTextForm.idAbbreviated
    Rem idTagTextForm.idVerbose
    .TagForm = idTagTextForm.idVerbose
End With
```

Do not assume that you are limited to exporting text using existing export filters. Because VBScript can write text files to disk, you can have your script traverse the text in a document and export it in any order you like, using whatever text-markup scheme you prefer. Here is a very simple example that shows how to export InCopy text as HTML (for the complete script, see ExportHTML):

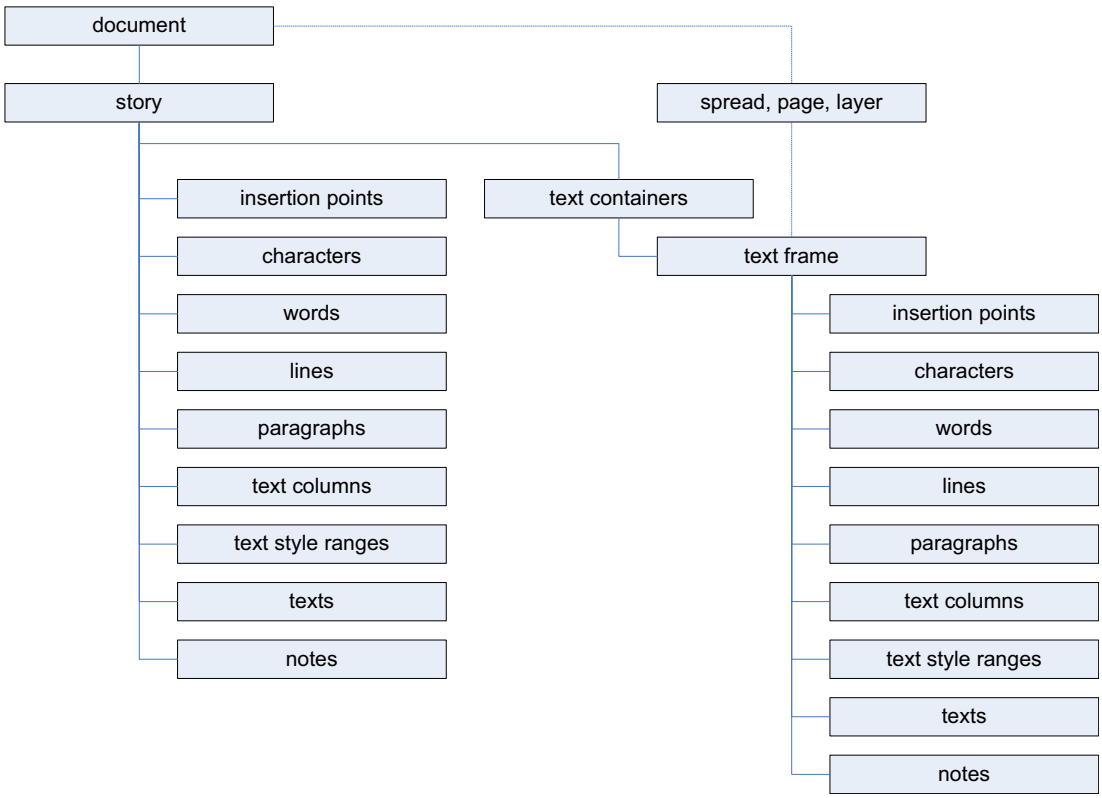
```
Function myExportHTML(myInCopy, myDocument)
    Rem Use the myStyleToTagMapping dictionary to set up
    Rem your paragraph style to tag mapping.
    Set myStyleToTagMapping = CreateObject("Scripting.Dictionary")
    Rem For each style to tag mapping, add a new item to the dictionary.
    myStyleToTagMapping.Add "body_text", "p"
    myStyleToTagMapping.Add "heading1", "h1"
    myStyleToTagMapping.Add "heading2", "h2"
    myStyleToTagMapping.Add "heading3", "h3"
    Rem End of style to tag mapping.
    If myDocument.Stories.Count <> 0 Then
        Rem Open a new text file.
        Set myDialog = CreateObject("UserAccounts.CommonDialog")
        myDialog.Filter = "HTML Files|*.html|All Files|*.*"
        myDialog.FilterIndex = 1
        myDialog.InitialDir = "C:\"
        myResult = myDialog.ShowOpen
        Rem If the user clicked the Cancel button, the result is null.
        If myResult = True Then
            myTextFileName = myDialog.FileName
            Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
            Set myTextFile = myFileSystemObject.CreateTextFile(myTextFileName)
```

```
For myCounter = 1 To myInCopy.Documents.Item(1).Stories.Count
    Set myStory = myDocument.Stories.Item(myCounter)
    For myParagraphCounter = 1 To myStory.Paragraphs.Count
        Set myParagraph = myStory.Paragraphs.Item(myParagraphCounter)
        If myParagraph.Tables.Count = 0 Then
            If myParagraph.TextStyleRanges.Count = 1 Then
                Rem If the paragraph is a simple paragraph--no tables,
                Rem no local formatting--then simply export the text of
                Rem the paragraph with the appropriate tag.
                myTag = myStyleToTagMapping.Item
                (myParagraph.AppliedParagraphStyle.Name)
                Rem If the tag comes back empty, map it to the basic
                Rem paragraph tag.
                If myTag = "" Then
                    myTag = "p"
                End If
                myStartTag = "<" & myTag & ">"
                myEndTag = "</" & myTag & ">"
                Rem If the paragraph is not the last paragraph
                Rem in the story, omit the return character.
                If myParagraph.Characters.Item(-1).Contents = vbCr Then
                    myString = myParagraph.Texts.ItemByRange
                    (myParagraph.Characters.Item(1),
                    myParagraph.Characters.Item(-2)).Item(1).Contents
                Else
                    myString = myParagraph.Contents
                End If
                Rem Write the paragraphs' text to the text file.
                myTextFile.WriteLine myStartTag & myString & myEndTag
            Else
                Rem Handle text style range export by iterating through
                the text style ranges in the paragraph..
                For myRangeCounter = 1 To
                myParagraph.TextStyleRanges.Length
                    myTextStyleRange =
                    myParagraph.TextStyleRanges.Item(myRangeCounter)
                    If myTextStyleRange.Characters.Item(-1) = vbCr Then
                        myString = myTextStyleRange.Texts.ItemByRange
                        (myTextStyleRange.Characters.Item(1),
                        myTextStyleRange.Characters.Item(-2)).Item(1).
                        Contents
                    Else
                        myString = myTextStyleRange.Contents
                    End If
                    Select Case myTextStyleRange.FontStyle
                        Case "Bold":
                            myString = "<b>" & myString & "</b>"
                        Case "Italic":
                            myString = "<i>" & myString & "</i>"
                    End Select
                    myTextFile.write myString
                Next
                myTextFile.write vbCr
            End If
        Else
            Rem Handle table export (assumes that there is only one
            table per paragraph, and that the table is in the paragraph
            by itself).
            Set myTable = myParagraph.Tables.Item(1)
            myTextFile.write "<table border = 1>"
```

```
                    For myRowCounter = 1 To myTable.Rows.Count
                        myTextFile.write "<tr>"
                        For myColumnCounter = 1 To myTable.Columns.Count
                            If myRowCounter = 1 Then
                                myString = "<th>" & myTable.Rows.Item
                                (myRowCounter).Cells.Item(myColumnCounter).
                                Texts.Item(1).Contents & "</th>"
                            Else
                                myString = "<td>" & myTable.Rows.Item
                                (myRowCounter).Cells.Item(myColumnCounter).
                                Texts.Item(1).Contents & "</td>"
                            End If
                            myTextFile.write myString
                        Next
                            myTextFile.WriteLine "</tr>"
                    Next
                    myTextFile.WriteLine "</table>"
                End If
            Next
            Rem Close the text file.
            myTextFile.Close
        Next
    End If
End If
End Function
```

# Text objects

The following diagram shows a view of InCopy's text-object model. There are two main types of text object: *layout* objects (text frames) and *text-stream* objects (stories, insertion points, characters, and words, for example). The diagram uses the natural-language terms for the objects; when you write scripts, you will use the corresponding terms from your scripting language:




For any text-stream object, the `parent` of the object is the story containing the object. To get a reference to the text frame (or text frames) containing a given text object, use the `ParentTextFrames` property.

For a text frame, the `parent` of the text frame usually is the page or spread containing the text frame. If the text frame is inside a group or was pasted inside another page item, the `parent` of the text frame is the containing page item. If the text frame was converted to an anchored frame, the `parent` of the text frame is the character containing the anchored frame.

## Selections

Usually, InCopy scripts act on a text selection. The following script shows how to determine the type of the current selection. Unlike many other sample scripts, this script does not actually do anything; it simply presents a selection filtering routine you can use in your own scripts. (For the complete script, see TextSelection.)

```
Set myInCopy = CreateObject("InCopy.Application")
If myInCopy.Documents.Count <> 0 Then
    Rem If the selection contains more than one item, the selection
    Rem is not text selected with the Type tool.
    If myInCopy.Selection.Count = 1 Then
        Select Case TypeName(myInCopy.Selection.Item(1))
        Case "InsertionPoint", "Character", "Word", "TextStyleRange", "Line",
        "Paragraph", "TextColumn", "Text"
            MsgBox "The selection is a text object."
            Rem A real script would now act on the text object
            Rem or pass it on to a function.
        Case Else
            MsgBox "The selected object is not a text object.
            Select some text and try again."
        End Select
    Else
         MsgBox "Please select some text and try again."
    End If
Else
    MsgBox "No documents are open. Please open a document,
    select some text, and try again."
End If
```

## Moving and copying text

To move a text object to another location in text, use the `move` method. To copy the text, use the `duplicate` method (which has exactly the same parameters as the `move` method). The following script fragment shows how it works (for the complete script, see MoveText):

```
Rem Create an example document.
Set myDocument = myInCopy.Documents.Add
Rem Create a series of paragraphs in the default story.
Set myStory = myDocument.Stories.Item(1)
myStory.Contents = "WordA" & vbcr & "WordB" & vbcr & "WordC" & vbcr & "WordD" & vbcr
Rem Move WordC before WordA.
myStory.Paragraphs.Item(3).Move idLocationOptions.idBefore,
myStory.Paragraphs.Item(1)
Rem Move WordB after WordD (into the same paragraph).
myStory.Paragraphs.Item(3).Move idLocationOptions.idAfter,
myStory.Paragraphs.Item(-1).Words.Item(1)
Rem Note that moving text removes it from its original location.
```

When you want to transfer formatted text from one document to another, you also can use the `move` method. Using the `move` or `duplicate` method is better than using copy and paste; to use copy and paste, you must make the document visible and select the text you want to copy. Using `move` or `duplicate` is much faster and more robust. The following script shows how to move text from one document to another using `move` and `duplicate` (for the complete script, see MoveTextBetweenDocuments):

```
Rem Moves formatted text from one document to another.
Rem Create an example document.
Set mySourceDocument = myInCopy.Documents.Add
Rem Add text to the default story.
Set mySourceStory = mySourceDocument.stories.item(1)
mySourceStory.Contents = "This is the source text." & vbCr & "This text is not the
source text." & vbcr
mySourceStory.paragraphs.item(1).pointSize = 24
Rem Create a new document to move the text to.
Set myTargetDocument = myInCopy.Documents.Add
Rem Create a text frame in the target document.
Set myTargetStory = myTargetDocument.stories.item(1)
myTargetStory.contents = "This is the target text. Insert the source text after this
paragraph." & vbcr
mySourceStory.paragraphs.item(1).duplicate idLocationOptions.idAfter,
myTargetStory.insertionPoints.item(-1)
```

One way to copy unformatted text from one text object to another is to get the `contents` property of a text object, then use that string to set the `contents` property of another text object. The following script shows how to do this (for the complete script, see CopyUnformattedText):

```
Rem Shows how to remove formatting from text as you
Rem move it to other locations in a document.
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Add
set myStory = myDocument.stories.item(1)
myStory.contents = "This is a formatted string." & vbcr & "Text pasted after this text
will retain its formatting." & vbcr & vbcr & "Text moved to the following line will take
on the formatting of the insertion point." & vbcr & "Italic: "
Rem Apply formatting to the first paragraph.
myStory.Paragraphs.Item(1).FontStyle = "Bold"
Rem Apply formatting to the last paragraph.
myStory.Paragraphs.Item(-1).FontStyle = "Italic"
Rem Copy from one frame to another using a simple copy.
myInCopy.Select myStory.Paragraphs.Item(1).Words.Item(1)
myInCopy.Copy
myInCopy.Select myStory.Paragraphs.Item(2).InsertionPoints.Item(-1)
myInCopy.Paste
Rem Copy the unformatted string from the first word to the end of the story
Rem by getting and setting the contents of text objects. Note that this doesn't
Rem really copy the text; it replicates the text string from one text location
Rem to another.
myStory.InsertionPoints.Item(-1).Contents =
myStory.Paragraphs.Item(1).Words.Item(1).contents
```

## Text objects and iteration

When your script moves, deletes, or adds text while iterating through a series of text objects, you can easily end up with invalid text references. The following script demonstrates this problem (for the complete script, see TextIterationWrong):

```
Set myDocument =  myInCopy.Documents.Add
Set myStory = myDocument.Stories.Item(1)
myString = "Paragraph 1." & vbCr & "Delete this paragraph." & vbCr & "Paragraph 2." &
vbCr & "Paragraph 3." & vbCr & "Paragraph 4." & vbCr & "Paragraph 5." & vbCr & "Delete
this paragraph." & vbCr & "Paragraph 6." & vbCr
myStory.Contents = myString
Rem The following for loop cause an error.
For myParagraphCounter = 1 to myStory.Paragraphs.Count
    If myStory.Paragraphs.Item(myParagraphCounter).Words.Item(1).contents = "Delete"
Then
        myStory.Paragraphs.Item(myParagraphCounter).Delete
    Else
        myStory.Paragraphs.Item(myParagraphCounter).PointSize = 24
    End If
Next
```

In the preceding example, some paragraphs are left unformatted. How does this happen? The loop in the script iterates through the paragraphs from the first paragraph in the story to the last. As it does so, it deletes paragraphs beginning with "Delete." When the script deletes the second paragraph, the third paragraph moves up to take its place. When the loop counter reaches 3, the script processes the paragraph that had been the fourth paragraph in the story; the original third paragraph is now the second paragraph and is skipped.

To avoid this problem, iterate backward through the text objects, as shown in the following script (from the TextIterationRight tutorial script):

```
Rem Shows how to iterate through text.
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Add
Set myStory = myDocument.Stories.Item(1)
myString = "Paragraph 1." & vbCr & "Delete this paragraph." & vbCr & "Paragraph 2." &
vbCr & "Paragraph 3." & vbCr & "Paragraph 4." & vbCr & "Paragraph 5." & vbCr & "Delete
this paragraph." & vbCr & "Paragraph 6." & vbCr
myStory.Contents = myString
Rem The following for loop will format all of the paragraphs by iterating
Rem backwards through the paragraphs in the story.
For myParagraphCounter = myStory.Paragraphs.Count To 1 Step -1
    If myStory.Paragraphs.Item(myParagraphCounter).Words.Item(1).contents = "Delete"
Then
        myStory.Paragraphs.Item(myParagraphCounter).Delete
    Else
        myStory.Paragraphs.Item(myParagraphCounter).PointSize = 24
    End If
Next
```

# Formatting text

In the previous sections of this chapter, we added text to a document and worked with stories and text objects. In this section, we apply formatting to text. All the typesetting capabilities of InCopy are available to scripting.

## Setting text defaults

You can set text defaults for both the application and each document. Text defaults for the application determine the text defaults in all new documents. Text defaults for a document set the formatting of all new text objects in that document. (For the complete script, see TextDefaults.)

```
Rem Sets the text defaults of a new document, which set the default formatting
Rem for all new text frames. Existing text frames are unaffected.
Set myInCopy = CreateObject("InCopy.Application")
myInCopy.ViewPreferences.HorizontalMeasurementUnits = idMeasurementUnits.idPoints
myInCopy.ViewPreferences.VerticalMeasurementUnits = idMeasurementUnits.idPoints
Rem To set the application text formatting defaults, replace "myInCopy"
Rem with a reference to a document in the following lines.
With myInCopy.TextDefaults
    .AlignToBaseline = True
    Rem Because the font might not be available, it's usually best
    Rem to trap errors using "On Error Resume Next" error handling.
    Rem Fill in the name of a font on your system.
    Err.Clear
    On Error Resume Next
    .AppliedFont = myInCopy.Fonts.Item("Minion Pro")
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    Rem Because the font style might not be available, it's usually best
    Rem to trap errors using "On Error Resume Next" error handling.
    Err.Clear
    On Error Resume Next
    .FontStyle = "Regular"
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0

    Rem Because the language might not be available, it's usually best
    Rem to trap errors using "On Error Resume Next" error handling.
    Err.Clear
    On Error Resume Next
    .AppliedLanguage = "English: USA"
    If Err.Number <> 0 Then
        Err.Clear
    End If
    On Error GoTo 0
    .AutoLeading = 100
    .BalanceRaggedLines = False
    .BaselineShift = 0
    .Capitalization = idCapitalization.idNormal
    .Composer = "Adobe Paragraph Composer"
    .DesiredGlyphScaling = 100
    .DesiredLetterSpacing = 0
    .DesiredWordSpacing = 100
    .DropCapCharacters = 0
    If .DropCapCharacters <> 0 Then
        .DropCapLines = 3
        Rem Assumes that application has a default character style named "myDropCap"
        .DropCapStyle = myInCopy.CharacterStyles.Item("myDropCap")
    End If
    .FillColor = myInCopy.Colors.Item("Black")
    .FillTint = 100
    .FirstLineIndent = 14
    .GridAlignFirstLineOnly = False
    .HorizontalScale = 100
    .HyphenateAfterFirst = 3
    .HyphenateBeforeLast = 4
    .HyphenateCapitalizedWords = False
```

```
.HyphenateLadderLimit = 1
.HyphenateWordsLongerThan = 5
.Hyphenation = True
.HyphenationZone = 36
.HyphenWeight = 9
.Justification = idJustification.idLeftAlign
.KeepAllLinesTogether = False
.KeepLinesTogether = True
.KeepFirstLines = 2
.KeepLastLines = 2
.KeepWithNext = 0
.KerningMethod = "Optical"
.Leading = 14
.LeftIndent = 0
.Ligatures = True
.MaximumGlyphScaling = 100
.MaximumLetterSpacing = 0
.MaximumWordSpacing = 160
.MinimumGlyphScaling = 100
.MinimumLetterSpacing = 0
.MinimumWordSpacing = 80
.NoBreak = False
.OTFContextualAlternate = True
.OTFDiscretionaryLigature = True
.OTFFigureStyle = idOTFFigureStyle.idProportionalOldstyle
.OTFFraction = True
.OTFHistorical = True
.OTFOrdinal = False
.OTFSlashedZero = True
.OTFSwash = False
.OTFTitling = False
.OverprintFill = False
.OverprintStroke = False
.PointSize = 11
.Position = idPosition.idNormal
.RightIndent = 0
.RuleAbove = False
If .RuleAbove = True Then
    .RuleAboveColor = myInCopy.Colors.Item("Black")
    .RuleAboveGapColor = myInCopy.Swatches.Item("None")
    .RuleAboveGapOverprint = False
    .RuleAboveGapTint = 100
    .RuleAboveLeftIndent = 0
    .RuleAboveLineWeight = 0.25
    .RuleAboveOffset = 14
    .RuleAboveOverprint = False
    .RuleAboveRightIndent = 0
    .RuleAboveTint = 100
    .RuleAboveType = myInCopy.StrokeStyles.Item("Solid")
    .RuleAboveWidth = idRuleWidth.idColumnWidth
End If
.RuleBelow = False
If .RuleBelow = True Then
    .RuleBelowColor = myInCopy.Colors.Item("Black")
    .RuleBelowGapColor = myInCopy.Swatches.Item("None")
    .RuleBelowGapOverPrint = False
    .RuleBelowGapTint = 100
    .RuleBelowLeftIndent = 0
    .RuleBelowLineWeight = 0.25
    .RuleBelowOffset = 0
```

```
        .RuleBelowOverPrint = False
        .RuleBelowRightIndent = 0
        .RuleBelowTint = 100
        .RuleBelowType = myInCopy.StrokeStyles.Item("Solid")
        .RuleBelowWidth = idRuleWidth.idColumnWidth
    End If
    .SingleWordJustification = idSingleWordJustification.idLeftAlign
    .Skew = 0
    .SpaceAfter = 0
    .SpaceBefore = 0
    .StartParagraph = idStartParagraph.idAnywhere
    .StrikeThru = False
    If .StrikeThru = True Then
        .StrikeThroughColor = myInCopy.Colors.Item("Black")
        .StrikeThroughGapColor = myInCopy.Swatches.Item("None")
        .StrikeThroughGapOverprint = False
        .StrikeThroughGapTint = 100
        .StrikeThroughOffset = 3
        .StrikeThroughOverprint = False
        .StrikeThroughTint = 100
        .StrikeThroughType = myInCopy.StrokeStyles.Item("Solid")
        .StrikeThroughWeight = 0.25
    End If
    .StrokeColor = myInCopy.Swatches.Item("None")
    .StrokeTint = 100
    .StrokeWeight = 0
    .Tracking = 0
    .Underline = False
    If .Underline = True Then
        .UnderlineColor = myInCopy.Colors.Item("Black")
        .UnderlineGapColor = myInCopy.Swatches.Item("None")
        .UnderlineGapOverprint = False
        .UnderlineGapTint = 100
        .UnderlineOffset = 3
        .UnderlineOverprint = False
        .UnderlineTint = 100
        .UnderlineType = myInCopy.StrokeStyles.Item("Solid")
        .UnderlineWeight = 0.25
    End If
    .VerticalScale = 100
End With
```

## Fonts

The fonts collection of an InCopy application object contains all fonts accessible to InCopy. By contrast, the fonts collection of a document contains only those fonts used in the document. The fonts collection of a document also contains any missing fonts—fonts used in the document that are not accessible to InCopy. The following script shows the difference between application fonts and document fonts (for the complete script, see FontCollections):

```
Rem Shows the difference between the fonts collection of the application
Rem and the fonts collection of a document.
Set myInCopy = CreateObject("InCopy.Application")
Set myApplicationFonts = myInCopy.Fonts
Set myDocument = myInCopy.Documents.Add
Set myStory = myDocument.Stories.Item(1)
myString = "Document Fonts:" & vbCr
For myCounter = 1 To myDocument.Fonts.Count
    myString = myString & myDocument.Fonts.Item(myCounter).Name & vbCr
Next
myString = myString & vbCr & "Application Fonts:" & vbCr
For myCounter = 1 To myInCopy.Fonts.Count
    myString = myString & myInCopy.Fonts.Item(myCounter) & vbCr
Next
myStory.Contents = myString
```

**NOTE:** Font names typically are of the form *familyName*<tab>*fontStyle*, where *familyName* is the name of the font family, <tab> is a tab character, and *fontStyle* is the name of the font style. For example:

```
"Adobe Caslon Pro<tab>Semibold Italic"
```

## Applying a font

To apply a local font change to a range of text, use the `appliedFont` property, as shown in the following script fragment (from the ApplyFont tutorial script):

```
Rem Given a reference to InCopy "myInCopy," a font name "myFontName"
Rem and a text object "myText"...
myText.appliedFont = myInCopy.Fonts.Item(myFontName)
```

You also can apply a font by specifying the font-family name and font style, as shown in the following script fragment:

```
myText.AppliedFont = myInCopy.Fonts.Item("Adobe Caslon Pro")
myText.FontStyle = "Semibold Italic"
```

## Changing text properties

Text objects in InCopy have literally dozens of properties corresponding to their formatting attributes. Even a single insertion point features properties that affect the formatting of text—up to and including properties of the paragraph containing the insertion point. The SetTextProperties tutorial script shows how to set every property of a text object. A fragment of the script follows:

```
Rem Shows how to set all read/write properties of a text object.
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Add()
Set myStory = myDocument.Stories.Item(1)
myStory.Contents = "x"
Set myTextObject = myStory.Characters.Item(1)
With myTextObject
    .AlignToBaseline = False
    .AppliedCharacterStyle = myDocument.CharacterStyles.Item("[None]")
    myFontName = "Minion Pro" & vbTab & "Regular"
    .AppliedFont = myInCopy.Fonts.Item(myFontName)
    .AppliedLanguage = myInCopy.LanguagesWithVendors.Item("English: USA")
    .AppliedNumberingList = myDocument.NumberingLists.Item("[Default]")
    .AppliedParagraphStyle = myDocument.ParagraphStyles.Item("[No Paragraph Style]")
    .AutoLeading = 120
    .BalanceRaggedLines = idBalanceLinesStyle.idNoBalancing
    .BaselineShift = 0
    .BulletsAlignment = idListAlignment.idLeftAlign
    .BulletsAndNumberingListType = idListType.idNoList
    .BulletsCharacterStyle = myDocument.CharacterStyles.Item("[None]")
    .BulletsTextAfter = "^t"
    .Capitalization = idCapitalization.idNormal
    .Composer = "Adobe Paragraph Composer"
    .DesiredGlyphScaling = 100
    .DesiredLetterSpacing = 0
    .DesiredWordSpacing = 100
    .DropCapCharacters = 0
    .DropCapLines = 0
    .DropCapStyle = myDocument.CharacterStyles.Item("[None]")
    .DropcapDetail = 0
    Rem More text properties in the tutorial script.
```

## Changing text color

You can apply colors to the fill and stroke of text characters, as shown in the following script fragment (from the TextColors tutorial script):

```
Rem Given two colors "myColorA" and "myColorB"...
set myStory = myDocument.Stories.Item(1)
myStory.contents = "Text" & vbCr & "Color"
Set myText = myStory.Paragraphs.Item(1)
myText.PointSize = 72
myText.Justification = idJustification.idCenterAlign
Set myText = myStory.Paragraphs.Item(2)
myText.StrokeWeight = 3
myText.PointSize = 144
myText.Justification = idJustification.idCenterAlign

Rem Apply a color to the fill of the text.
Set myText = myStory.Paragraphs.Item(1)
myText.FillColor = myDocument.Colors.Item("DGC1_446a")
Rem Use the itemByRange method to apply the color to the stroke of the text.
myText.StrokeColor = myDocument.Swatches.Item("DGC1_446b")
Set myText = myStory.Paragraphs.Item(2)
myText.FillColor = myDocument.Swatches.Item("DGC1_446b")
myText.StrokeColor = myDocument.Swatches.Item("DGC1_446a")
myText.StrokeWeight = 3
```

## Creating and applying styles

While you can use scripting to apply local formatting—as in some of the examples earlier in this chapter—you probably will want to use character and paragraph styles to format your text. Using styles creates a link between the formatted text and the style, which makes it easier to redefine the style, collect the text formatted with a given style, or find and/or change the text. Paragraph and character styles are key to text-formatting productivity and should be a central part of any script that applies text formatting.

The following script fragment shows how to create and apply paragraph and character styles (for the complete script, see CreateStyles):

```
Rem Shows how to create and apply a paragraph style and a character style.
Set myInCopy = CreateObject("InCopy.Application")
Rem Create an example document.
Set myDocument = myInCopy.Documents.Add
Rem Create a color for use by one of the paragraph styles we'll create.
Set myColor = myAddColor(myDocument, "Red", idColorModel.idProcess, Array(0, 100, 100,
0))
Rem Create a text frame on page 1.
Set myStory = myDocument.Stories.Item(1)
Rem Fill the text frame with placeholder text.
myStory.Contents = "Normal text. Text with a character style applied to it. More normal
text."
Rem Create a character style named "myCharacterStyle" if
Rem no style by that name already exists.
Set myCharacterStyle = myAddStyle(myDocument, "myCharacterStyle", 1)
Rem At this point, the variable myCharacterStyle contains a reference to a character
Rem style object, which you can now use to specify formatting.
myCharacterStyle.FillColor = myColor
Rem Create a paragraph style named "myParagraphStyle" if
Rem no style by that name already exists.
Set myParagraphStyle = myAddStyle(myDocument, "myParagraphStyle", 2)
Rem At this point, the variable myParagraphStyle contains a reference to a
Rem paragraph-style object, which you can now use to specify formatting.
myStory.Texts.Item(1).ApplyParagraphStyle myParagraphStyle, True
Set myStartCharacter = myStory.Characters.Item(14)
Set myEndCharacter = myStory.Characters.Item(55)
Set myText = myStory.Texts.ItemByRange(myStartCharacter, myEndCharacter)
myText.Item(1).ApplyCharacterStyle myCharacterStyle
Function myAddColor(myDocument, myColorName, myColorModel, myColorValue)
    On Error Resume Next
    Set myColor = myDocument.Colors.Item(myColorName)
    If Err.Number <> 0 Then
        Set myColor = myDocument.Colors.Add
        myColor.Name = myColorName
    End If
    Err.Clear
    On Error GoTo 0
    myColor.Model = myColorModel
    myColor.ColorValue = myColorValue
    Set myAddColor = myColor
End Function
Function myAddStyle(myDocument, myStyleName, myStyleType)
    On Error Resume Next
    Select Case myStyleType
        Case 1:
            Set myStyle = myDocument.CharacterStyles.Item(myStyleName)
            If Err.Number <> 0 Then
                Set myStyle = myDocument.CharacterStyles.Add
```

```
            myStyle.Name = myStyleName
        End If
        Err.Clear
        On Error GoTo 0
    Case 2:
        Set myStyle = myDocument.ParagraphStyles.Item(myStyleName)
        If Err.Number <> 0 Then
            Set myStyle = myDocument.ParagraphStyles.Add
            myStyle.Name = myStyleName
        End If
        Err.Clear
        On Error GoTo 0
    Case 3:
        Set myStyle = myDocument.ObjectStyles.Item(myStyleName)
        If Err.Number <> 0 Then
            Set myStyle = myDocument.ObjectStyles.Add
            myStyle.Name = myStyleName
        End If
        Err.Clear
        On Error GoTo 0
End Select
Set myAddStyle = myStyle
End Function
```

Why use the `ApplyParagraphStyle` method instead of setting the `AppliedParagraphStyle` property of the text object? The method gives the ability to override existing formatting; setting the property to a style retains local formatting.

Why check for the existence of a style when creating a new document? It always is possible that the style exists as an application default style. If it does, trying to create a new style with the same name results in an error.

Nested styles apply character-style formatting to a paragraph according to a pattern. The following script fragment shows how to create a paragraph style containing nested styles (for the complete script, see NestedStyles):

```
Rem At this point, the variable myParagraphStyle contains a reference to a
Rem paragraoh-style object, which you can now use to specify formatting.
Set myNestedStyle = myParagraphStyle.NestedStyles.Add
myNestedStyle.AppliedCharacterStyle =
myDocument.CharacterStyles.Item("myCharacterStyle")
myNestedStyle.Delimiter = "."
myNestedStyle.Inclusive = True
myNestedStyle.Repetition = 1
Set myStartCharacter = myStory.Characters.Item(1)
Set myEndCharacter = myStory.Characters.Item(-1)
Rem Use the ItemByRange method to apply the paragraph to all text in the
Rem story. Note the story object does not have the applyStyle method.)
Set myText = myStory.Texts.ItemByRange(myStartCharacter, myEndCharacter).Item(1)
myText.ApplyParagraphStyle myParagraphStyle, True
```

## Deleting a style

When you delete a style using the user interface, you can choose how you want to format any text tagged with that style. InCopy scripting works the same way, as shown in the following script fragment (from the RemoveStyle tutorial script):

```
Rem Delete the paragraph style myParagraphStyleA and replace with myParagraphStyleB.
myParagraphStyleA.Delete myDocument.ParagraphStyles.Item("myParagraphStyleB")
```

## Importing paragraph and character styles

You can import paragraph and character styles from other InCopy documents. The following script fragment shows how (for the complete script, see ImportTextStyles):

```
Rem Import the styles from the saved document.
Rem ImportStyles parameters:
Rem Format as idImportFormat enumeration. Options for text styles are:
Rem    idImportFormat.idParagraphStylesFormat
Rem    idImportFormat.idCharacterStylesFormat
Rem    idImportFormat.idTextStylesFormat
Rem From as string (file path)
Rem GlobalStrategy as idGlobalClashResolutionStrategy enumeration. Options are:
Rem    idGlobalClashResolutionStrategy.idDoNotLoadTheStyle
Rem    idGlobalClashResolutionStrategy.idLoadAllWithOverwrite
Rem    idGlobalClashResolutionStrategy.idLoadAllWithRename
myNewDocument.ImportStyles idImportFormat.idTextStylesFormat, "c:\styles.icml",
idGlobalClashResolutionStrategy.idLoadAllWithOverwrite
```

# Finding and changing text

The find/change feature is one of the most powerful InCopy tools for working with text. It is fully supported by scripting, and scripts can use find/change to go far beyond what can be done using the InCopy user interface. InCopy has three ways of searching for text:

- You can find text and text formatting and change it to other text and/or text formatting. This type of find and change operation uses the `FindTextPreferences` and `ChangeTextPreferences` objects to specify parameters for the `findText` and `changeText` methods.
- You can find text using regular expressions, or "grep." This type of find and change operation uses the `FindGrepPreferences` and `ChangeGrepPreferences` objects to specify parameters for the `findGrep` and `changeGrep` methods.
- You can find specific glyphs (and their formatting) and replace them with other glyphs and formatting. This type of find and change operation uses the`FindGlyphPreferences` and `ChangeGlyphPreferences` objects to specify parameters for the `findGlyph` and `changeGlyph` methods.

All find and change methods take a single optional parameter, `ReverseOrder`, which specifies the order in which the results of the search are returned. If you are processing the results of a find or change operation in a way that adds or removes text from a story, you might face the problem of invalid text references, as discussed in "Text objects and iteration" on page 43. In this case, you can either construct your loops to iterate backward through the collection of returned text objects, or you can have the search operation return the results in reverse order and then iterate through the collection normally.

## Find/change preferences

Before searching for text, you probably will want to clear find and change preferences, to make sure the settings from previous searches have no effect on your search. You also need to set a few find and change preferences to specify the text, formatting, regular expression, or glyph you want to find and/or change. A typical find/change operation involves the following steps:

1. Clear the find/change preferences. Depending on the type of find/change operation, this can take one of the following three forms:

```
Rem Find/Change text preferences (where "myInCopy" is a
Rem reference to the InCopy application
myInCopy.FindTextPreferences = idNothingEnum.idNothing
myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
Rem Find/Change grep preferences
myInCopy.FindGrepPreferences = idNothingEnum.idNothing
myInCopy.ChangeGrepPreferences = idNothingEnum.idNothing
Rem Find/Change glyph preferences
myInCopy.FindGlyphPreferences = idNothingEnum.idNothing
myInCopy.ChangeGlyphPreferences = idNothingEnum.idNothing
```

2. Set up find/change parameters.
3. Execute the find/change operation.
4. Clear find/change preferences again.

## Finding text

The following script fragment shows how to find a specified string of text. While the script fragment searches the entire document, you also can search stories, text frames, paragraphs, text columns, or any other text object. The `findText` method and its parameters are the same for all text objects. (For the complete script, see FindText.)

```
Rem Clear the find/change preferences.
myInCopy.FindTextPreferences = idNothingEnum.idNothing
myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
Rem Search the document for the string "Text".
myInCopy.FindTextPreferences.FindWhat = "text"
Rem Set the find options.
myInCopy.FindChangeTextOptions.CaseSensitive = False
myInCopy.FindChangeTextOptions.IncludeFootnotes = False
myInCopy.FindChangeTextOptions.IncludeHiddenLayers = False
myInCopy.FindChangeTextOptions.IncludeLockedLayersForFind = False
myInCopy.FindChangeTextOptions.IncludeLockedStoriesForFind = False
myInCopy.FindChangeTextOptions.IncludeMasterPages = False
myInCopy.FindChangeTextOptions.WholeWord = False
Set myFoundItems = myInCopy.Documents.Item(1).FindText()
MsgBox ("Found " & CStr(myFoundItems.Count) & " instances of the search string.")
myInCopy.FindTextPreferences = idNothingEnum.idNothing
myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
```

The following script fragment shows how to find a specified string of text and replace it with a different string (for the complete script, see ChangeText):

```
Rem Clear the find/change preferences.
myInCopy.FindTextPreferences = idNothingEnum.idNothing
myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
Rem Search the document for the string "Text".
myInCopy.FindTextPreferences.FindWhat = "text"
Rem Set the find options.
myInCopy.FindChangeTextOptions.CaseSensitive = False
myInCopy.FindChangeTextOptions.IncludeFootnotes = False
myInCopy.FindChangeTextOptions.IncludeHiddenLayers = False
myInCopy.FindChangeTextOptions.IncludeLockedLayersForFind = False
myInCopy.FindChangeTextOptions.IncludeLockedStoriesForFind = False
myInCopy.FindChangeTextOptions.IncludeMasterPages = False
myInCopy.FindChangeTextOptions.WholeWord = False
Set myFoundItems = myInCopy.Documents.Item(1).FindText
MsgBox ("Found " & CStr(myFoundItems.Count) & " instances of the search string.")
myInCopy.FindTextPreferences = idNothingEnum.idNothing
myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
```

## Finding and changing formatting

To find and change text formatting, you set other properties of the `findTextPreferences` and `changeTextPreferences` objects, as shown in the following script fragment (from the FindChangeFormatting tutorial script):

```
Rem Clear the find/change preferences.
myInCopy.FindTextPreferences = idNothingEnum.idNothing
myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
Rem Set the find options.
myInCopy.FindChangeTextOptions.CaseSensitive = False
myInCopy.FindChangeTextOptions.IncludeFootnotes = False
myInCopy.FindChangeTextOptions.IncludeHiddenLayers = False
myInCopy.FindChangeTextOptions.IncludeLockedLayersForFind = False
myInCopy.FindChangeTextOptions.IncludeLockedStoriesForFind = False
myInCopy.FindChangeTextOptions.IncludeMasterPages = False
myInCopy.FindChangeTextOptions.WholeWord = False
Rem Search the document for the 24 point text and change it to 10 point text.
myInCopy.FindTextPreferences.PointSize = 24
myInCopy.ChangeTextPreferences.PointSize = 10
Set myFoundItems = myInCopy.Documents.Item(1).ChangeText
myInCopy.FindTextPreferences = idNothingEnum.idNothing
myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
```

You also can search for a string of text and apply formatting, as shown in the following script fragment (from the FindChangeStringFormatting tutorial script):

```
Rem Clear the find/change preferences before the search.
myInCopy.FindTextPreferences = idNothingEnum.idNothing
myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
Rem Set the general find/change options.
myInCopy.findChangeTextOptions.caseSensitive = false
myInCopy.findChangeTextOptions.includeFootnotes = false
myInCopy.findChangeTextOptions.includeHiddenLayers = false
myInCopy.findChangeTextOptions.includeLockedLayersForFind = false
myInCopy.findChangeTextOptions.includeLockedStoriesForFind = false
myInCopy.findChangeTextOptions.includeMasterPages = false
myInCopy.findChangeTextOptions.wholeWord = false
Rem The following line will only work if your default
Rem font has a font style named "Bold" if not, change
Rem the text to a font style used by your default font.
myInCopy.ChangeTextPreferences.FontStyle = "Bold"
Rem In this example, we'll use the InCopy search
Rem metacharacter "^9" to find any digit.
myInCopy.FindTextPreferences.FindWhat = "WIDGET^9^9^9^9"
set myFoundItems = myDocument.ChangeText
MsgBox ("Changed " & CStr(myFoundItems.Count) & " instances of the search string.")
Rem Clear the find/change preferences after the search.
myInCopy.FindTextPreferences = idNothingEnum.idNothing
myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
```

## Using grep

InCopy supports regular expression find/change through the `findGrep` and `changeGrep` methods. Regular-expression find and change also can find text with a specified format or replace text formatting with formatting specified in the properties of the `changeGrepPreferences` object. The following script fragment shows how to use these methods and the related preferences objects (for the complete script, see FindGrep):

```
Rem Clear the find/change preferences.
myInCopy.FindGrepPreferences = idNothingEnum.idNothing
myInCopy.ChangeGrepPreferences = idNothingEnum.idNothing
Rem Set the find options.
myInCopy.FindChangeGrepOptions.IncludeFootnotes = False
myInCopy.FindChangeGrepOptions.IncludeHiddenLayers = False
myInCopy.FindChangeGrepOptions.IncludeLockedLayersForFind = False
myInCopy.FindChangeGrepOptions.IncludeLockedStoriesForFind = False
myInCopy.FindChangeGrepOptions.IncludeMasterPages = False
Rem Regular expression for finding an email address.
myInCopy.FindGrepPreferences.FindWhat = "(?i)[A-Z]*?@[A-Z]*?[.]..."
Rem Apply the change to 24-point text only.
myInCopy.FindGrepPreferences.PointSize = 24
myInCopy.ChangeGrepPreferences.Underline = True
myInCopy.Documents.Item(1).ChangeGrep
Rem Clear the find/change preferences after the search.
myInCopy.FindGrepPreferences = idNothingEnum.idNothing
myInCopy.ChangeGrepPreferences = idNothingEnum.idNothing
```

**NOTE:** The `findChangeGrepOptions` object lacks two properties of the `FindChangeTextOptions` object: `WholeWord`and `CaseSensitive`. This is because you can set these options using the regular expression string itself. Use `(?i)` to turn case sensitivity on and `(?-i)` to turn case sensitivity off. Use `\>` to match the beginning of a word and `\<` to match the end of a word, or use `\b` to match a word boundary.

One handy use for grep find/change is to convert text markup (that is, some form of tagging plain text with formatting instructions) into InCopy formatted text. PageMaker paragraph tags (which are not the

same as PageMaker tagged text-format files) are an example of a simplified text-markup scheme. In a text file marked up using this scheme, paragraph style names appear at the start of a paragraph, as shown in these examples:

```
<heading1>This is a heading.

<body_text>This is body text.
```

We can create a script that uses grep find in conjunction with text find/change operations to apply formatting to the text and remove the markup tags, as shown in the following script fragment (from the ReadPMTags tutorial script):

```
Function myReadPMTags(myInCopy, myStory)
    Set myDocument = myStory.Parent
    Rem Reset the findGrepPreferences to ensure that previous settings
    Rem do not affect the search.
    myInCopy.FindGrepPreferences = idNothingEnum.idNothing
    myInCopy.ChangeGrepPreferences = idNothingEnum.idNothing
    myInCopy.FindGrepPreferences.findWhat = "(?i)^<\s*\w+\s*>"
    Set myFoundItems = myStory.findGrep
    If myFoundItems.Count <> 0 Then
        Set myFoundTags = CreateObject("Scripting.Dictionary")
        For myCounter = 1 To myFoundItems.Count
            If Not (myFoundTags.Exists(myFoundItems.Item(myCounter).Contents)) Then
            myFoundTags.Add myFoundItems.Item(myCounter).Contents,
                myFoundItems.Item(myCounter).Contents
            End If
        Next
        Rem At this point, we have a list of tags to search for.
        For Each myFoundTag In myFoundTags
            myString = myFoundTag
            Rem Find the tag using findWhat.
            myInCopy.FindTextPreferences.findWhat = myString
            Rem Extract the style name from the tag.
            myStyleName = Mid(myString, 2, Len(myString) - 2)
            Rem Create the style if it does not already exist.
            Set myStyle = myAddStyle(myDocument, myStyleName)
            Rem Apply the style to each instance of the tag.
            myInCopy.ChangeTextPreferences.AppliedParagraphStyle = myStyle
            myStory.ChangeText
            Rem Reset the changeTextPreferences.
            myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
            Rem Set the changeTo to an empty string.
            myInCopy.ChangeTextPreferences.ChangeTo = ""
            Rem Search to remove the tags.
            myStory.ChangeText
            Rem Reset the find/change preferences again.
            myInCopy.ChangeTextPreferences = idNothingEnum.idNothing
        Next
```

```
    End If
    Rem Reset the findGrepPreferences.
    myInCopy.FindGrepPreferences = idNothingEnum.idNothing
End Function
Function myAddStyle(myDocument, myStyleName)
    On Error Resume Next
    Set myStyle = myDocument.ParagraphStyles.Item(myStyleName)
    If Err.Number <> 0 Then
        Set myStyle = myDocument.ParagraphStyles.Add
        myStyle.Name = myStyleName
    End If
    Err.Clear
    On Error GoTo 0
    Set myAddStyle = myStyle
End Function
```

## Using glyph search

You can find and change individual characters in a specific font using the `FindGlyph` and `ChangeGlyph` methods and the associated `FindGlyphPreferences` and `ChangeGlyphPreferences` objects. The following scripts fragment shows how to find and change a glyph in a sample document (for the complete script, see FindChangeGlyphs):

```
Rem Clear the find/change preferences.
myInCopy.FindGlyphPreferences = idNothingEnum.idNothing
myInCopy.ChangeGlyphPreferences = idNothingEnum.idNothing
Rem Set the find options.
myInCopy.FindChangeGrepOptions.IncludeFootnotes = False
myInCopy.FindChangeGrepOptions.IncludeHiddenLayers = False
myInCopy.FindChangeGrepOptions.IncludeLockedLayersForFind = False
myInCopy.FindChangeGrepOptions.IncludeLockedStoriesForFind = False
myInCopy.FindChangeGrepOptions.IncludeMasterPages = False
Rem You must provide a font that is used in the document for the
Rem AppliedFont property of the FindGlyphPreferences object.
myInCopy.FindGlyphPreferences.AppliedFont = myDocument.Fonts.Item("Minion Pro
Regular");
Rem Provide the glyph ID, not the glyph Unicode value.
myInCopy.FindGlyphPreferences.GlyphID = 500;
Rem The appliedFont of the changeGlyphPreferences object can be
Rem any font available to the application.
myInCopy.changeGlyphPreferences.AppliedFont = myInCopy.Fonts.Item("Times New Roman
Regular");
myInCopy.Documents.Item(1).ChangeGlyph
Rem Clear the find/change preferences after the search.
myInCopy.FindGlyphPreferences = idNothingEnum.idNothing
myInCopy.ChangeGlyphPreferences = idNothingEnum.idNothing
```

# Tables

Tables can be created from existing text using the `ConvertTextToTable` method, or an empty table can be created at any insertion point in a story. The following script fragment shows three different ways to create a table (for the complete script, see MakeTable):

```
Set myStory = myDocument.Stories.Item(1)
Set myStartCharacter = myStory.Paragraphs.Item(7).Characters.Item(1)
Set myEndCharacter = myStory.Paragraphs.Item(7).Characters.Item(-2)
Set myText = myStory.Texts.ItemByRange(myStartCharacter, myEndCharacter).Item(1)
Rem The convertToTable method takes three parameters:
Rem [ColumnSeparator as string]
Rem [RowSeparator as string]
Rem [NumberOfColumns as integer] (only used if the ColumnSeparator
Rem and RowSeparator values are the same)
Rem In the last paragraph in the story, columns are separated by commas
Rem and rows are separated by semicolons, so we provide those characters
Rem to the method as parameters.
Set myTable = myText.ConvertToTable(",", ";")
Set myStartCharacter = myStory.Paragraphs.Item(2).Characters.Item(1)
Set myEndCharacter = myStory.Paragraphs.Item(5).Characters.Item(-2)
Set myText = myStory.Texts.ItemByRange(myStartCharacter, myEndCharacter).Item(1)
Rem In the second through the fifth paragraphs, colums are separated by
Rem tabs and rows are separated by returns. These are the default delimiter
Rem parameters, so we don't need to provide them to the method.
Set myTable = myText.ConvertToTable
Rem You can also explicitly add a table--you don't have to convert text to a table.
Set myTable = myStory.InsertionPoints.Item(-1).Tables.Add
myTable.ColumnCount = 3
myTable.BodyRowCount = 3
```

The following script fragment shows how to merge table cells (for the complete script, see MergeTableCells):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Add
Set myStory = myDocument.Stories.Item(1)
myString = "Table" & vbCr
myStory.Contents = myString
Set myTable = myStory.InsertionPoints.Item(-1).Tables.Add
myTable.ColumnCount = 4
myTable.BodyRowCount = 4
Rem Merge all of the cells in the first column.
myTable.Cells.Item(1).Merge myTable.Columns.Item(1).Cells.Item(-1)
Rem Convert column 2 into 2 cells (rather than 4).
myTable.Columns.Item(2).Cells.Item(-1).Merge myTable.Columns.Item(2).Cells.Item(-2)
myTable.Columns.Item(2).Cells.Item(1).Merge myTable.Columns.Item(2).Cells.Item(2)
Rem Merge the last two cells in row 1.
myTable.Rows.Item(1).Cells.Item(-1).Merge myTable.Rows.Item(1).Cells.Item(-1)
Rem Merge the last two cells in row 3.
myTable.Rows.Item(3).Cells.Item(-2).Merge myTable.Rows.Item(3).Cells.Item(-1)
```

The following script fragment shows how to split table cells (for the complete script, see SplitTableCells):

```
myTable.Cells.Item(1).Split idHorizontalOrVertical.idHorizontal
myTable.Columns.Item(1).Split idHorizontalOrVertical.idVertical
myTable.Cells.Item(1).Split idHorizontalOrVertical.idVertical
myTable.Rows.Item(-1).Split idHorizontalOrVertical.idHorizontal
myTable.Cells.Item(-1).Split idHorizontalOrVertical.idVertical
For myRowCounter = 1 To myTable.Rows.Count
    Set myRow = myTable.Rows.Item(myRowCounter)
    For myCellCounter = 1 To myRow.Cells.Count
        myString = "Row: " & myRowCounter & " Cell: " & myCellCounter
        myRow.Cells.Item(myCellCounter).contents = myString
    Next
Next
```

The following script fragment shows how to create header and footer rows in a table (for the complete script, see HeaderAndFooterRows):

```
Set myTable = myDocument.Stories.Item(1).Tables.Item(1)
Rem Convert the first row to a header row.
myTable.Rows.Item(1).RowType = idRowTypes.idHeaderRow
Rem Convert the last row to a footer row.
myTable.Rows.Item(-1).RowType = idRowTypes.idFooterRow
```

The following script fragment shows how to apply formatting to a table (for the complete script, see TableFormatting):

```
Set myTable = myStory.Tables.Item(1)
Rem Convert the first row to a header row.
myTable.Rows.Item(1).RowType = idRowTypes.idHeaderRow
Rem Use a reference to a swatch, rather than to a color.
myTable.Rows.Item(1).FillColor = myDocument.Swatches.Item("DGC1_446b")
myTable.Rows.Item(1).FillTint = 40
myTable.Rows.Item(2).FillColor = myDocument.Swatches.Item("DGC1_446a")
myTable.Rows.Item(2).FillTint = 40
myTable.Rows.Item(3).FillColor = myDocument.Swatches.Item("DGC1_446a")
myTable.Rows.Item(3).FillTint = 20
myTable.Rows.Item(4).FillColor = myDocument.Swatches.Item("DGC1_446a")
myTable.Rows.Item(4).FillTint = 40
Rem Iterate through the cells to apply the cell stroke formatting.
For myCounter = 1 To myTable.Cells.Count
    myTable.Cells.Item(myCounter).TopEdgeStrokeColor =
myDocument.Swatches.Item("DGC1_446b")
    myTable.Cells.Item(myCounter).TopEdgeStrokeWeight = 1
    myTable.Cells.Item(myCounter).BottomEdgeStrokeColor =
myDocument.Swatches.Item("DGC1_446b")
    myTable.Cells.Item(myCounter).BottomEdgeStrokeWeight = 1
    Rem When you set a cell stroke to a swatch, make certain you also set the
    Rem stroke weight.
    myTable.Cells.Item(myCounter).LeftEdgeStrokeColor =
myDocument.Swatches.Item("None")
    myTable.Cells.Item(myCounter).LeftEdgeStrokeWeight = 0
    myTable.Cells.Item(myCounter).RightEdgeStrokeColor =
myDocument.Swatches.Item("None")
    myTable.Cells.Item(myCounter).RightEdgeStrokeWeight = 0
Next
```

The following script fragment shows how to add alternating row formatting to a table (for the complete script, see AlternatingRows):

```
Set myTable = myDocument.stories.Item(1).tables.Item(1)
Rem Convert the first row to a header row.
myTable.rows.Item(1).rowType = idRowTypes.idHeaderRow
Rem Applly alternating fills to the table.
myTable.alternatingFills = idAlternatingFillsTypes.idAlternatingRows
myTable.startRowFillColor = myDocument.swatches.Item("DGC1_446a")
myTable.startRowFillTint = 60
myTable.endRowFillColor = myDocument.swatches.Item("DGC1_446b")
myTable.endRowFillTint = 50
```

The following script fragment shows how to process the selection when text or table cells are selected. In this example, the script displays an alert for each selection condition, but a real production script would then do something with the selected item(s). (For the complete script, see TableSelection.)

```
If myInCopy.Documents.Count <> 0 Then
    If myInCopy.Selection.Count <> 0 Then
        Select Case TypeName(myInCopy.Selection.Item(1))
            Rem When a row, a column, or a range of cells is selected,
            Rem the type returned is "Cell"
            Case "Cell"
                MsgBox ("A cell is selected.")
            Case "Table"
                MsgBox ("A table is selected.")
            Case "InsertionPoint", "Character", "Word", "TextStyleRange",
            "Line", "Paragraph", "TextColumn", "Text"
                If TypeName(myInCopy.Selection.Item(1).Parent) = "Cell" Then
                    MsgBox ("The selection is inside a table cell.")
                Else
                    MsgBox ("The selection is not inside a table.")
                End If
            Case Else
                MsgBox ("The selection is not inside a table.")
        End Select
    End If
End If
```

# Autocorrect

The autocorrect feature can correct text as you type. The following script shows how to use it (for the complete script, see Autocorrect):

```
Rem The autocorrect preferences object turns the autocorrect feature
Rem on or off.
ReDim myNewWordPairList(0)
Set myInCopy = CreateObject("InCopy.Application")
Rem Add a word pair to the autocorrect list. Each AutoCorrectTable is linked
Rem to a specific language.
Set myAutoCorrectTable = myInCopy.AutoCorrectTables.Item("English: USA")
Rem To safely add a word pair to the auto correct table, get the current
Rem word pair list, then add the new word pair to that array, and then
Rem set the autocorrect word pair list to the array.
myWordPairList = myAutoCorrectTable.AutoCorrectWordPairList
ReDim myNewWordPairList(UBound(myWordPairList) + 1)
For myCounter = 0 To UBound(myWordPairList) - 1
    myNewWordPairList(myCounter) = myWordPairList(myCounter)
Next
Rem Add a new word pair to the array.
myNewWordPairList(UBound(myNewWordPairList)) = (Array("paragarph", "paragraph"))
Rem Update the word pair list.
myAutoCorrectTable.AutoCorrectWordPairList = myNewWordPairList
Rem To clear all autocorrect word pairs in the current dictionary:
Rem myAutoCorrectTable.autoCorrectWordPairList = array(())
Rem Turn autocorrect on if it's not on already.
If myInCopy.AutoCorrectPreferences.AutoCorrect = False Then
    yInCopy.AutoCorrectPreferences.AutoCorrect = True
End If
myInCopy.AutoCorrectPreferences.AutoCorrectCapitalizationErrors = True
```

# Footnotes

The following script fragment shows how to add footnotes to a story (for the complete script, see Footnotes):

```
Set myDocument = myInCopy.Documents.Item(1)
With myDocument.FootnoteOptions
    .SeparatorText = vbTab
    .MarkerPositioning = idFootnoteMarkerPositioning.idSuperscriptMarker
End With
Set myStory = myDocument.Stories.Item(1)
Rem Add four footnotes at random locations in the story.
For myCounter = 1 To 4
    myRandomNumber = CLng(myGetRandom(1, myStory.Words.Count))
    Set myWord = myStory.Words.Item(myRandomNumber)
    Set myFootnote = myWord.InsertionPoints.Item(-1).Footnotes.Add
    Rem Note: when you create a footnote, it contains text--the footnote
    Rem marker and the separator text (if any). If you try to set the text of
    Rem the footnote by setting the footnote contents, you will delete the
    Rem marker. Instead, append the footnote text, as shown below.
    myFootnote.InsertionPoints.Item(-1).Contents = "This is a footnote."
Next
Rem This function gets a random number in the range myStart to myEnd.
Function myGetRandom(myStart, myEnd)
       Rem Here's how to generate a random number from a given range:
       Rem Int((upperbound - lowerbound + 1) * Rnd + lowerbound)
       myGetRandom = (myEnd - (myStart + 1)) * Rnd + myStart
End Function
```

# 5 User Interfaces

| Chapter Update Status | |
|---|---|
| CS6 | Unchanged |

VBScript can create dialog boxes for simple yes/no questions and text entry, but you probably will need to create more complex dialog boxes for your scripts. InCopy scripting can add dialog boxes and can populate them with common user-interface controls, like pop-up lists, text-entry fields, and numeric-entry fields. If you want your script to collect and act on information entered by you or any other user of your script, use the `dialog` object.

This chapter shows how to work with InCopy dialog scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

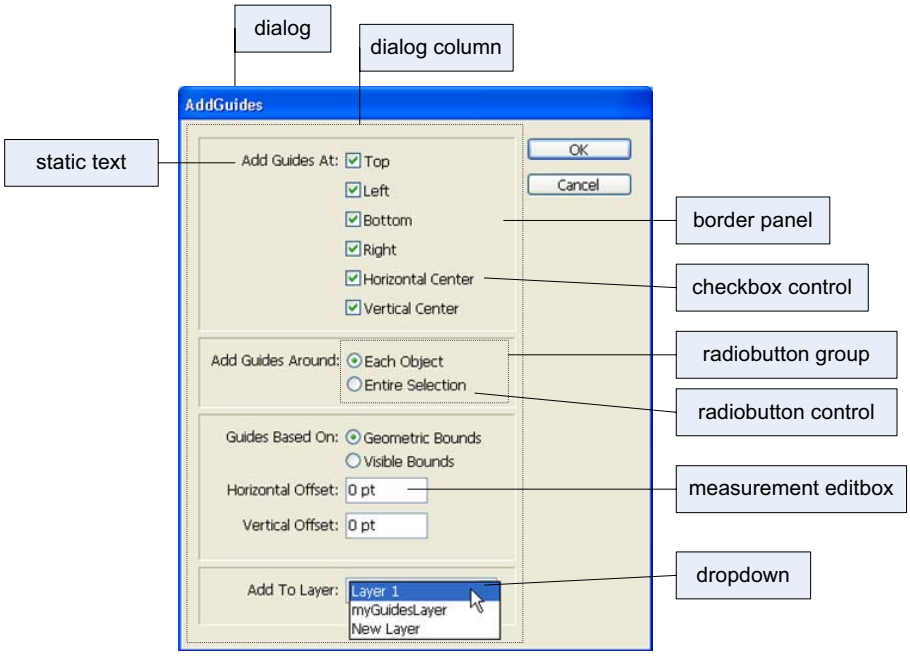
**NOTE:** InCopy scripts written in JavaScript also can include user interfaces created using the Adobe *ScriptUI* component. This chapter includes some ScriptUI scripting tutorials; for more information, see *Adobe Creative Suite® 3 JavaScript Tools Guide*.

**NOTE:** Although Visual Basic applications can create complete user interfaces, they run from a separate Visual Basic executable file. InCopy scripting includes the ability to create complex dialog boxes that appear inside InCopy and look very much like the program's standard user interface. VBScripts that run from the Scripts palette are much faster than scripts run from an external application.

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script.

## Dialog-box overview

An InCopy dialog box is an object like any other InCopy scripting object. The dialog box can contain several different types of elements (known collectively as "widgets"), as shown in the following figure:




The items in the figure are defined in the following table:

| Dialog-box element | InCopy name |
|---|---|
| Text-edit fields | Text editbox control |
| Numeric-entry fields | Real editbox, integer editbox, measurement editbox, percent editbox, angle editbox |
| Pop-up menus | Drop-down control |
| Control that combines a text-edit field with a pop-up menu | Combo-box control |
| Check box | Check-box control |
| Radio buttons | Radio-button control |

The `dialog` object itself does not directly contain the controls; that is the purpose of the `DialogColumn` object. `DialogColumns` give you a way to control the positioning of controls within a dialog box. Inside `DialogColumns`, you can further subdivide the dialog box into other `DialogColumns` or `BorderPanels` (both of which can, if necessary, contain more `DialogColumns` and `BorderPanels`).

Like any other InCopy scripting object, each part of a dialog box has its own properties. For example, a `CheckboxControl` has a property for its text (`StaticLabel`) and another property for its state (`CheckedState`). The `Dropdown` control has a property (`StringList`) for setting the list of options that appears on the control's menu.

To use a dialog box in your script, create the `dialog` object, populate it with various controls, display the dialog box, and then gather values from the dialog-box controls to use in your script. Dialog boxes remain in InCopy's memory until they are destroyed. This means you can keep a dialog box in memory and have data stored in its properties used by multiple scripts, but it also means the dialog boxes take up memory

and should be disposed of when they are not in use. In general, you should destroy a dialog-box object before your script finishes executing.

# Your first InCopy dialog box

The process of creating an InCopy dialog box is very simple: add a dialog box, add a dialog column to the dialog box, and add controls to the dialog column. The following script demonstrates the process (for the complete script, see SimpleDialog).

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDialog = myInCopy.Dialogs.Add
myDialog.name = "Simple Dialog"
Rem Add a dialog column.
With myDialog.DialogColumns.Add
    With .StaticTexts.Add
        .StaticLabel = "This is a very simple dialog box."
    End With
End With
Rem Show the dialog box.
myResult = myDialog.Show
Rem If the user clicked OK, display one message;
Rem if they clicked Cancel, display a different message.
If myResult = True Then
    MsgBox "You clicked the OK button."
Else
    MsgBox "You clicked the Cancel button."
End If
Rem Remove the dialog box from memory.
myDialog.Destroy
```

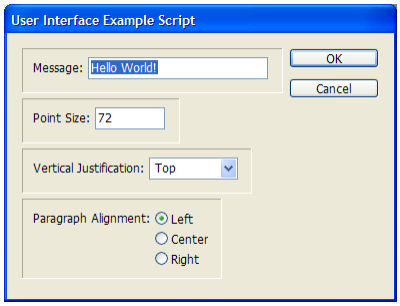
# Adding a user interface to “Hello World”

In this example, we add a simple user interface to the Hello World tutorial script presented in Chapter 2, “Getting Started.” The options in the dialog box provide a way for you to specify the sample text and change the point size of the text. For the complete script, see HelloWorldUI.

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDialog = myInCopy.Dialogs.Add
myDialog.CanCancel = True
myDialog.Name = " Simple User Interface Example Script"
Set myDialogColumn = myDialog.DialogColumns.Add
Set myTextEditField = myDialogColumn.TextEditboxes.Add
myTextEditField.EditContents = "Hello World!"
myTextEditField.MinWidth = 180
Rem Create a number (real) entry field.
Set myPointSizeField = myDialogColumn.RealEditboxes.Add
myPointSizeField.EditValue = 72
myDialog.Show
Rem Get the values from the dialog box controls.
myString = myTextEditField.EditContents
myPointSize = myPointSizeField.EditValue
Rem Remove the dialog box from memory.
myDialog.Destroy
Rem Create a new document.
Set myDocument = myInCopy.Documents.Add
Set myStory = myDocument.Stories.Item(1)
Rem Enter the text from the dialog box.
myStory.Contents = myString
Rem Set the size of the text to the size you entered in the dialog box.
myStory.Texts.Item(1).PointSize = myPointSize
```

# Creating a more complex user interface

In the next example, we add more controls and different types of controls to the sample dialog box. The example creates a dialog box that resembles the following:




For the complete script, see ComplexUI.

```
Function myDisplayDialog(myInCopy)
    ReDim mySwatchNames(myInCopy.Swatches.Count -1)
    For myCounter = 1 To myInCopy.Swatches.Count
        Set mySwatch = myInCopy.Swatches.Item(myCounter)
        mySwatchNames(myCounter - 1) = mySwatch.Name
    Next
    Set myDialog = myInCopy.Dialogs.Add
    myDialog.CanCancel = True
    myDialog.Name = "ComplexUI"
    Rem Create a dialog column.
    With myDialog.DialogColumns.Add
        Rem Create a border panel.
        With .BorderPanels.Add
            With .DialogColumns.Add
                With .DialogRows.Add
                    With .StaticTexts.Add
                        .StaticLabel = "Message:"
                    End With
                    Set myTextEditField = .TextEditboxes.Add
                    myTextEditField.EditContents = "Hello World!"
                    myTextEditField.MinWidth = 180
                End With
            End With
        End With
        With .BorderPanels.add
            With .DialogColumns.Add
                With .DialogRows.Add
                    With .StaticTexts.Add
                        .StaticLabel = "Point Size:"
                    End With
                    Set myPointSizeField = .MeasurementEditboxes.Add
                    myPointSizeField.EditUnits = idMeasurementUnits.idPoints
                    myPointSizeField.EditValue = 72
                End With
            End With
        End with
        With .BorderPanels.Add
            With .DialogColumns.Add
                With .DialogRows.Add
                    With .StaticTexts.Add
                        .StaticLabel = "Paragraph Alignment:"
                    End With
                    Set myRadioButtonGroup = .RadiobuttonGroups.Add
                    With myRadioButtonGroup
                        With .RadiobuttonControls.Add
                            .StaticLabel = "Left"
                            .CheckedState = True
                        End With
                        With .RadiobuttonControls.Add
                            .StaticLabel = "Center"
                        End With
                        With .RadiobuttonControls.Add
                            .StaticLabel = "Right"
                        End With
                    End With
                End With
            End With
        End With
        With .BorderPanels.Add
            With .DialogColumns.Add
```

```
                With .DialogRows.Add
                    With .StaticTexts.Add
                        .StaticLabel = "Text Color:"
                    End With
                    Set mySwatchDropdown = .Dropdowns.Add
                    mySwatchDropdown.StringList = mySwatchNames
                    mySwatchDropdown.SelectedIndex = 2
                End With
            End With
        End With
    End With
    Rem If the user clicked OK, then create the example document.
    If myDialog.Show = True Then
        Rem Get the values from the dialog box controls.
        myString = myTextEditField.EditContents
        myPointSize = myPointSizeField.EditValue
        myParagraphAlignment = myRadioButtonGroup.SelectedButton
        mySwatchName = mySwatchNames(mySwatchDropdown.SelectedIndex)
        myDialog.Destroy
        myCreateExampleDocument myInCopy, myString, myPointSize,
        myParagraphAlignment, mySwatchName
    Else
        myDialog.Destroy
    End If
End Function
Function myCreateExampleDocument(myInCopy, myString, myPointSize,
myParagraphAlignment, mySwatchName)
    Set myDocument = myInCopy.Documents.Add
    Set myStory = myDocument.Stories.Item(1)
    Rem Enter the text from the dialog box in the story.
    myStory.Contents = myString
    Rem Set the size of the text to the size you entered in the dialog box.
    myStory.Texts.Item(1).PointSize = myPointSize
    Rem set the paragraph alignment of the text to the
    Remdialog radio button choice.
    Select Case myParagraphAlignment
        Case 0
            myStory.Texts.Item(1).Justification = idJustification.idLeftAlign
        Case 1
            myStory.Texts.Item(1).Justification = idJustification.idCenterAlign
        Case Else
            myStory.Texts.Item(1).Justification = idJustification.idRightAlign
    End Select
    Rem Apply the selected swatch to the fill of the text.
    myStory.Texts.Item(1).FillColor = myDocument.Swatches.Item(mySwatchName)
End function
```

# Working with ScriptUI

JavaScripts can make, create, and define user-interface elements using an Adobe scripting component named ScriptUI. ScriptUI gives script writers a way to create floating palettes, progress bars, and interactive dialog boxes that are far more complex than InCopy's built-in `dialog` object.

This does not mean, however, that user-interface elements written using Script UI are not accessible to VBScript users. InCopy scripts can execute scripts written in other scripting languages using the DoScript method.

## Creating a progress bar with ScriptUI

The following sample script shows how to create a progress bar using JavaScript and ScriptUI, then how to use the progress bar from an VBScript (for the complete script, see ProgressBar):

```
#targetengine "session"
var myProgressPanel;
var myMaximumValue = 300;
var myProgressBarWidth = 300;
var myIncrement = myMaximumValue/myProgressBarWidth;
myCreateProgressPanel(myMaximumValue, myProgressBarWidth);
function myCreateProgressPanel(myMaximumValue, myProgressBarWidth){
    myProgressPanel = new Window('window', 'Progress');
    with(myProgressPanel){
        myProgressPanel.myProgressBar = add('progressbar', [12, 12, myProgressBarWidth,
24], 0, myMaximumValue);
    }
}
```

The following script fragment shows how to call the progress bar created in the preceding script using a VBScript (for the complete script, see CallProgressBar):

```
Set myInCopy = CreateObject("InCopy.Application")
myString = "myProgressPanel = myCreateProgressPanel(100, 400);" & vbcr
myString = myString & "myProgressPanel.show();" & vbcr
myInCopy.DoScript myString, idScriptLanguage.idJavascript
For myCounter = 1 to 100
    Set myStory = myInCopy.Documents.Item(1)
    myStory.InsertionPoints.Item(-1).Contents = "x"
    myString = "myProgressPanel.myProgressBar.value = " & cstr(myCounter) &
    "/myIncrement;" & vbcr
    myInCopy.DoScript myString, idScriptLanguage.idJavascript
    If(myCounter = 100) Then
        myString = "myProgressPanel.myProgressBar.value = 0;" & vbcr
        myString = myString & "myProgressPanel.hide();" & vbcr
        myInCopy.DoScript myString, idScriptLanguage.idJavascript
    End If
Next
```

## Creating a button-bar panel with ScriptUI

If you want to run your scripts by clicking buttons in a floating palette, you can create one using JavaScript and ScriptUI. It does not matter which scripting language the scripts themselves use.

The following tutorial script shows how to create a simple floating panel. The panel can contain a series of buttons, with each button being associated with a script stored on disk. Click the button, and the panel runs the script (the script, in turn, can display dialog boxes or other user-interface elements). The button in the panel can contain text or graphics. (For the complete script, see ButtonBar.)

The tutorial script reads an XML file in the following form:

```
<buttons>
    <button>
        <buttonType></buttonType>
        <buttonName></buttonName>
        <buttonFileName></buttonFileName>
        <buttonIconFile></buttonIconFile>
    </button>
    ...
</buttons>
```

For example:

```
<buttons>
    <button>
        <buttonType>text</buttonType>
        <buttonName>FindChangeByList</buttonName>
        <buttonFileName>/c/buttons/FindChangeByList.jsx</buttonFileName>
        <buttonIconFile></buttonIconFile>
    </button>
    <button>
        <buttonType>text</buttonType>
        <buttonName>SortParagraphs</buttonName>
        <buttonFileName>/c/buttons/SortParagraphs.jsx</buttonFileName>
        <buttonIconFile></buttonIconFile>
    </button>
</buttons>
```

The following functions read the XML file and set up the button bar:

```
#targetengine "session"
var myButtonBar;
main();
function main(){
    myButtonBar = myCreateButtonBar();
    myButtonBar.show();
}
function myCreateButtonBar(){
    var myButtonName, myButtonFileName, myButtonType, myButtonIconFile, myButton;
    var myButtons = myReadXMLPreferences();
    if(myButtons != ""){
        myButtonBar = new Window('window', 'Script Buttons', undefined,
        {maximizeButton:false, minimizeButton:false});
        with(myButtonBar){
            spacing = 0;
            margins = [0,0,0,0];
            with(add('group')){
                spacing = 2;
                orientation = 'row';
                for(var myCounter = 0; myCounter < myButtons.length(); myCounter++){
                    myButtonName = myButtons[myCounter].xpath("buttonName");
                    myButtonType = myButtons[myCounter].xpath("buttonType");
                    myButtonFileName = myButtons[myCounter].xpath("buttonFileName");
                    myButtonIconFile = myButtons[myCounter].xpath("buttonIconFile");
                    if(myButtonType == "text"){
                        myButton = add('button', undefined, myButtonName);
                    }
                    else{
                        myButton = add('iconbutton', undefined,
                        File(myButtonIconFile));
                    }
                    myButton.scriptFile = myButtonFileName;
```

```
                myButton.onClick = function(){
                    myButtonFile = File(this.scriptFile)
                    app.doScript(myButtonFile);
                }
            }
        }
    }
    return myButtonBar;
}
function myReadXMLPreferences(){
    myXMLFile = File.openDialog("Choose the file containing your
    button bar defaults");
    var myResult = myXMLFile.open("r", undefined, undefined);
    var myButtons = "";
    if(myResult == true){
        var myXMLDefaults = myXMLFile.read();
        myXMLFile.close();
        var myXMLDefaults = new XML(myXMLDefaults);
        var myButtons = myXMLDefaults.xpath("/buttons/button");
    }
    return myButtons;
}
```

# 6 Menus

| Chapter Update Status | |
|---|---|
| CS6 | Unchanged |

InCopy scripting can add menu items, remove menu items, perform any menu command, and attach scripts to menu items.

This chapter shows how to work with InCopy menu scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

We assume that you have read Chapter 2, "Getting Started" and know how to create, install, and run a script.

## Understanding the menu model

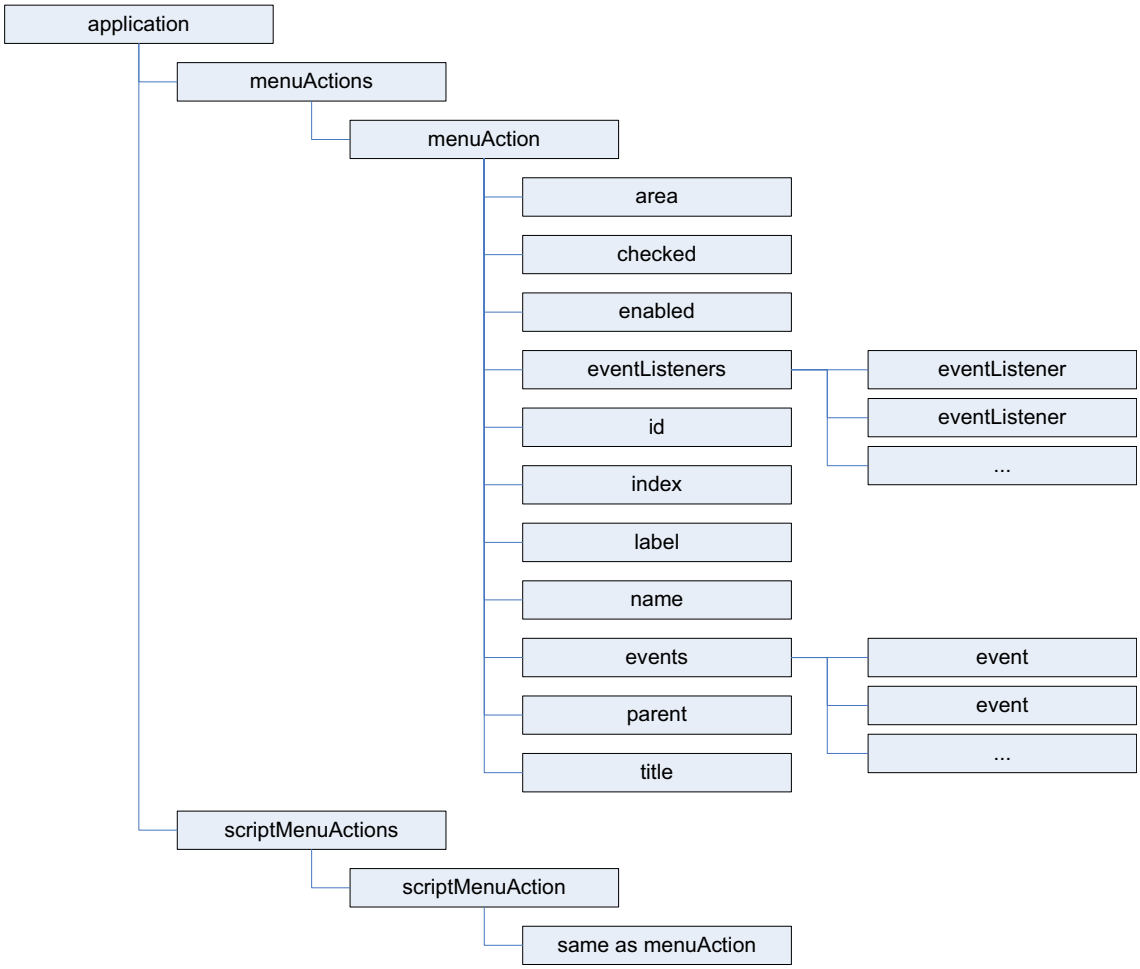
The InCopy menu-scripting model is made up of a series of objects that correspond to the menus you see in the application's user interface, including menus associated with panels as well as those displayed on the main menu bar. A `menu` object contains the following objects:

- `MenuItems` — The menu options shown on a menu. This does not include submenus.
- `MenuSeparators` — Lines used to separate menu options on a menu.
- `Submenus` — Menu options that contain further menu choices.
- `MenuElements` — All `MenuItems`, `MenuSeparators` and `Submenus` shown on a menu.
- `EventListeners` — These respond to user (or script) actions related to a menu.
- `Events` — The `events` triggered by a menu.

Every `MenuItem` is connected to a `MenuAction` through the `AssociatedMenuAction` property. The properties of the `MenuAction` define what happens when the menu item is chosen. In addition to the `MenuActions` defined by the user interface, InCopy scripters can create their own, `ScriptMenuActions`, which associate a script with a menu selection.

A `MenuAction` or `ScriptMenuAction` can be connected to zero, one, or more `MenuItems`.

The following diagram shows how the different menu objects relate to each other:




To create a list (as a text file) of all visible menu actions, run the following script fragment (from the GetMenuActions tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
Rem You'll need to fill in a valid file path on your system.
Set myTextFile = myFileSystemObject.CreateTextFile("c:\menuactions.txt", True, False)
For myCounter = 1 To myInCopy.MenuActions.Count
    Set myMenuAction = myInCopy.MenuActions.Item(myCounter)
    myTextFile.WriteLine myMenuAction.name
Next
myTextFile.Close
MsgBox "done!"
```

To create a list (as a text file) of all available menus, run the following script fragment (for the complete script listing, refer to the GetMenuNames tutorial script). Note that these scripts can be very slow, as there are a large number of menu names in InCopy.

```
Set myInCopy = CreateObject("InCopy.Application")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
Set myTextFile = myFileSystemObject.CreateTextFile("c:\menunames.txt", True, False)
For myMenuCounter = 1 To myInCopy.Menus.Count
    Set myMenu = myInCopy.Menus.Item(myMenuCounter)
    myTextFile.WriteLine myMenu.Name
    myProcessMenu myMenu, myTextFile
Next
myTextFile.Close
MsgBox "done!"
Function myProcessMenu(myMenuItem, myTextFile)
    myString = ""
    myMenuName = myMenuItem.Name
    For myCounter = 1 To myMenuItem.MenuElements.Count
        If TypeName(myMenuItem.MenuElements.Item(myCounter)) <>
        "MenuSeparator" Then
            myString = myGetIndent(myMenuItem.MenuElements.Item(myCounter),
            myString, False)
            myTextFile.WriteLine myString &
            myMenuItem.MenuElements.Item(myCounter).Name
            myMenuElementName = myMenuItem.MenuElements.Item(myCounter).Name
            myString = ""
            If TypeName(myMenuItem.MenuElements.Item(myCounter)) = "Submenu" Then
                If myMenuItem.MenuElements.Count > 0 Then
                    myProcessMenu myMenuItem.MenuElements.Item(myCounter),
                    myTextFile
                End If
            End If
        End If
    Next
End Function
Function myGetIndent(myMenuItem, myString, myDone)
    Do While myDone = False
        If TypeName(myMenuItem.Parent) = "Application" Then
            myDone = True
        Else
            myString = myString & vbTab
            myGetIndent myMenuItem.Parent, myString, myDone
        End If
    Loop
    myGetIndent = myString
End Function
```

## Localization and menu names

in InCopy scripting, `MenuItems`, `Menus`, `MenuActions`, and `Submenus` are all referred to by name. Because of this, scripts need a method of locating these objects that is independent of the installed locale of the application. To do this, you can use an internal database of strings that refer to a specific item, regardless of the locale. For example, to get the locale-independent name of a menu action, you can use the following script fragment (for the complete script, see GetKeyStrings):

```
Set myInCopy = CreateObject("InCopy.Application")
Rem Fill in the name of the menu action you want.
Set myMenuAction = myInCopy.MenuActions.Item("$ID/Convert to Note")
myKeyStrings = myInCopy.FindKeyStrings(myMenuAction.Name)
myString = ""
For Each myKeyString In myKeyStrings
    myString = myString & myKeyString & vbCr
Next
MsgBox myString
```

**NOTE:** It is much better to get the locale-independent name of a `MenuAction` than of a `Menus`, `MenuItem`, or `Submenu`, because the title of a `MenuAction` is more likely to be a single string. Many of the other menu objects return multiple strings when you use the `GetKeyStrings` method.

Once you have the locale-independent string you want to use, you can include it in your scripts. Scripts that use these strings will function properly in locales other than that of your version of InCopy.

To translate a locale-independent string into the current locale, use the following script fragment (from the TranslateKeyString tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
Rem Fill in the appropriate key string in the following line.
myString = myInCopy.TranslateKeyString("$ID/Convert to Note")
MsgBox myString
```

# Running a menu action from a script

Any of InCopy's built-in `MenuActions` can be run from a script. The `MenuAction` does not need to be attached to a `MenuItem`; however, in every other way, running a `MenuItem` from a script is exactly the same as choosing a menu option in the user interface. If selecting the menu option displays a dialog box, running the corresponding `MenuAction` from a script also displays a dialog box.

The following script shows how to run a `MenuAction` from a script (for the complete script, see InvokeMenuAction):

```
Set myInCopy = CreateObject("InCopy.Application")
Rem Get a reference to a menu action.
Set myMenuAction = myInCopy.MenuActions.Item("$ID/Convert to Note")
Rem Run the menu action. The example action will fail if you do not
Rem have text selected.
myMenuAction.Invoke
```

**NOTE:** In general, you should not try to automate InCopy processes by scripting menu actions and user-interface selections; InCopy's scripting object model provides a much more robust and powerful way to work. Menu actions depend on a variety of user-interface conditions, like the selection and the state of the window. Scripts using the object model work with the objects in an InCopy document directly, which means they do not depend on the user interface; this, in turn, makes them faster and more consistent.

# Adding menus and menu items

Scripts also can create new menus and menu items or remove menus and menu items, just as you can in the InCopy user interface. The following sample script shows how to duplicate the contents of a submenu to a new menu in another menu location (for the complete script, see CustomizeMenu):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myMainMenu = myInCopy.Menus.Item("Main")
Set myTypeMenu = myMainMenu.MenuElements.Item("Type")
Set myFontMenu = myTypeMenu.MenuElements.Item("Font")
Set myKozukaMenu = myFontMenu.Submenus.Item("Kozuka Mincho Pro ")
Set mySpecialFontMenu = myMainMenu.Submenus.Add("Kozuka Mincho Pro")
For myCounter = 1 To myKozukaMenu.MenuItems.Count
    Set myAssociatedMenuAction =
myKozukaMenu.MenuItems.Item(myCounter).AssociatedMenuAction
    mySpecialFontMenu.MenuItems.Add myAssociatedMenuAction
Next
```

To remove the custom menu added by the preceding script, run the RemoveSpecialFontMenu script.

```
Set myMainMenu = myInCopy.Menus.Item("Main")
Set mySpecialFontMenu = myMainMenu.Submenus.Item("Kozuka Mincho Pro")
mySpecialFontMenu.Delete
```

# Menus and events

Menus and submenus generate events as they are chosen in the user interface, and `MenuActions` and `ScriptMenuActions` generate events as they are used. Scripts can install `EventListeners` to respond to these events. The following table shows the events for the different menu scripting components:

| Object | Event | Description |
|---|---|---|
| `Menu` | `beforeDisplay` | Runs the attached script before the contents of the menu is shown. |
| `MenuAction` | `afterInvoke` | Runs the attached script when the associated `MenuItem` is selected, but after the `onInvoke` event. |
| | `beforeInvoke` | Runs the attached script when the associated `MenuItem` is selected, but before the `onInvoke` event. |
| `ScriptMenuAction` | `afterInvoke` | Runs the attached script when the associated `MenuItem` is selected, but after the `onInvoke` event. |
| | `beforeInvoke` | Runs the attached script when the associated `MenuItem` is selected, but before the `onInvoke` event. |
| | `beforeDisplay` | Runs the attached script before an internal request for the enabled/checked status of the `ScriptMenuActionScriptMenuAction`. |
| | `onInvoke` | Runs the attached script when the `ScriptMenuAction` is invoked. |
| `Submenu` | `beforeDisplay` | Runs the attached script before the contents of the `Submenu` are shown. |

For more about `Events` and `EventListeners`, see Chapter 7, "Events."

To change the items displayed in a menu, add an `EventListener` for the `beforeDisplay Event`. When the menu is selected, the `EventListener` can then run a script that enables or disables menu items, changes

the wording of menu item, or performs other tasks related to the menu. This mechanism is used internally to change the menu listing of available fonts, recent documents, or open windows.

# Working with script menu actions

You can use `ScriptMenuAction` to create a new `MenuAction` whose behavior is implemented through the script registered to run when the `onInvoke Event` is triggered.

The following script shows how to create a `ScriptMenuAction` and attach it to a menu item (for the complete script, see MakeScriptMenuAction). This script simply displays an alert when the menu item is selected.

```
Set myInCopy = CreateObject("InCopy.Application")
Set mySampleScriptAction = myInCopy.ScriptMenuActions.Add("Display Message")
Set myEventListener = mySampleScriptAction.EventListeners.Add("onInvoke",
"c:\message.vbs")
Set mySampleScriptMenu = myInCopy.Menus.Item("$ID/Main").Submenus.Add("Script Menu
Action")
Set mySampleScriptMenuItem = mySampleScriptMenu.MenuItems.Add(mySampleScriptAction)
```

The script file `message.vbs` contains the following code:

```
MsgBox("You selected an example script menu action.")
```

To remove the `Menu`, `Submenu`, `MenuItem`, and `ScriptMenuAction` created by the preceding script, run the following script fragment (from the RemoveScriptMenuAction tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
Set mySampleScriptAction = myInCopy.ScriptMenuActions.Item("Display Message")
mySampleScriptAction.Delete
Set mySampleScriptMenu = myInCopy.Menus.Item("$ID/Main").Submenus.Item("Script Menu
Action")
mySampleScriptMenu.Delete
```

You also can remove all `ScriptMenuAction`, as shown in the following script fragment (from the RemoveAllScriptMenuActions tutorial script). This script also removes the menu listings of the `ScriptMenuAction`, but it does not delete any menus or submenus you might have created.

```
Set myInCopy = CreateObject("InCopy.Application")
For myCounter = myInCopy.ScriptMenuActions.Count To 1 Step -1
    myInCopy.ScriptMenuActions.Item(myCounter).Delete
Next
```

You can create a list of all current `ScriptMenuActions`, as shown in the following script fragment (from the GetScriptMenuActions tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
Rem You'll need to fill in a valid file path for your system.
Set myTextFile = myFileSystemObject.CreateTextFile("c:\scriptmenuactionnames.txt",
True, False)
For myCounter = 1 To myInCopy.ScriptMenuActions.Count
    Set myScriptMenuAction = myInCopy.ScriptMenuActions.Item(myMenuCounter)
    myTextFile.WriteLine myScriptMenuAction.Name
Next
myTextFile.Close
```

`ScriptMenuAction` also can run scripts during their `beforeDisplay Event`, in which case they are executed before an internal request for the state of the `ScriptMenuAction` (for example, when the menu

item is about to be displayed). Among other things, the script can then change the menu names and/or set the enabled/checked status.

In the following sample script, we add an `EventListener` to the `beforeDisplay Event` that checks the current selection. If there is no selection, the script in the `EventListener` disables the menu item. If an item is selected, the menu item is enabled, and choosing the menu item displays the type of the first item in the selection. (For the complete script, see BeforeDisplay.)

```
Set myInCopy = CreateObject("InCopy.Application")
Set mySampleScriptAction = myInCopy.ScriptMenuActions.Add("Display Message")
Set myEventListener = mySampleScriptAction.EventListeners.Add("onInvoke",
"c:\WhatIsSelected.vbs ")
Set mySampleScriptMenu = myInCopy.Menus.Item("$ID/Main").Submenus.Add("Script Menu
Action")
Set mySampleScriptMenuItem = mySampleScriptMenu.MenuItems.Add(mySampleScriptAction)
mySampleScriptMenu.EventListeners.Add "beforeDisplay", "c:\BeforeDisplayHandler.vbs"
```

The BeforeDisplayHander tutorial script file contains the following script:

```
Set myInCopy = CreateObject("InCopy.Application")
Set mySampleScriptAction = myInCopy.ScriptMenuActions.Item("Display Message")
If myInCopy.Selection.Count > 0 Then
    mySampleScriptAction.Enabled = True
Else
    mySampleScriptAction.Enabled = False
End If
```

The WhatIsSelected tutorial script file contains the following script:

```
Set myInCopy = CreateObject("InCopy.Application")
myString = TypeName(myInCopy.Selection.Item(1))
MsgBox "The first item in the selection is a " & myString & "."
```

# 7 Events

| Chapter Update Status | |
|---|---|
| CS6 | Unchanged |

InCopy scripting can respond to common application and document events, like opening a file, creating a new file, printing, and importing text and graphic files from disk. In InCopy scripting, the `event` object responds to an event that occurs in the application. Scripts can be attached to events using the `EventListener` scripting object. Scripts that use events are the same as other scripts—the only difference is that they run automatically, as the corresponding event occurs, rather than being run by the user (from the Scripts palette).

This chapter shows how to work with InCopy event scripting. The sample scripts in this chapter are presented in order of complexity, starting with very simple scripts and building toward more complex operations.

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script.

This chapter covers application and document events. For a discussion of events related to menus, see Chapter 6, "Menus."

The InCopy event scripting model is similar to the Worldwide Web Consortium (W3C) recommendation for Document Object Model Events. For more information, see http://www.w3c.org.

## Understanding the event scripting model

The InCopy event scripting model is made up of a series of objects that correspond to the events that occur as you work with the application. The first object is the `event`, which corresponds to one of a limited series of actions in the InCopy user interface (or corresponding actions triggered by scripts).

To respond to an event, you register an `EventListener` with an object capable of receiving the event. When the specified event reaches the object, the `EventListener` executes the script function defined in its handler function (a reference to a script file on disk).

The following table shows a list of events to which `EventListeners` can respond. These events can be triggered by any available means, including menu selections, keyboard shortcuts, or script actions.

| User-Interface event | Event name | Description | Object type |
|---|---|---|---|
| Any menu action | `beforeDisplay` | Appears before the menu or submenu is displayed. | Event |
| | `beforeDisplay` | Appears before the script menu action is displayed or changed. | Event |
| | `beforeInvoke` | Appears after the menu action is chosen but before the content of the menu action is executed. | Event |
| | `afterInvoke` | Appears after the menu action is executed. | Event |
| | `onInvoke` | Executes the menu action or script menu action. | Event |
| Close | `beforeClose` | Appears after a close document request is made but before the document is closed. | DocumentEvent |
| | `afterClose` | Appears after a document is closed. | DocumentEvent |
| Export | `beforeExport` | Appears after an export request is made but before the document or page item is exported. | ImportExportEvent |
| | `afterExport` | Appears after a document or page item is exported. | ImportExportEvent |
| Import | `beforeImport` | Appears before a file is imported but before the incoming file is imported into a document (before place). | ImportExportEvent |
| | `afterImport` | Appears after a file is imported but before the file is placed on a page. | ImportExportEvent |
| New | `beforeNew` | Appears after a new document request but before the document is created. | DocumentEvent |
| | `afterNew` | Appears after a new document is created. | DocumentEvent |
| Open | `beforeOpen` | Appears after an open document request but before the document is opened. | DocumentEvent |
| | `afterOpen` | Appears after a document is opened. | DocumentEvent |
| Print | `beforePrint` | Appears after a print document request is made but before the document is printed. | DocumentEvent |
| | `afterPrint` | Appears after a document is printed. | DocumentEvent |

| User-Interface event | Event name | Description | Object type |
|---|---|---|---|
| Revert | `beforeRevert` | Appears after a document revert request is made but before the document is reverted to an earlier saved state. | DocumentEvent |
| | `afterRevert` | Appears after a document is reverted to an earlier saved state. | DocumentEvent |
| Save | `beforeSave` | Appears after a save document request is made but before the document is saved. | DocumentEvent |
| | `afterSave` | Appears after a document is saved. | DocumentEvent |
| Save A Copy | `beforeSaveACopy` | Appears after a document save-a-copy-as request is made but before the document is saved. | DocumentEvent |
| | `afterSaveACopy` | Appears after a document is saved. | DocumentEvent |
| Save As | `beforeSaveAs` | Appears after a document save-as request is made but before the document is saved. | DocumentEvent |
| | `afterSaveAs` | Appears after a document is saved. | DocumentEvent |

## About event properties and event propagation

When an action—whether initiated by a user or by a script—triggers an event, the event can spread, or *propagate*, through the scripting objects capable of responding to the event. When an event reaches an object that has an `EventListener` registered for that event, the `EventListener` is triggered by the event. An event can be handled by more than one object as it propagates.

There are three types of event propagation:

- **None —** Only the `EventListeners` registered to the event target are triggered by the event. The `beforeDisplay` event is an example of an event that does not propagate.
- **Capturing** — The event starts at the top of the scripting object model—the application—then propagates through the model to the target of the event. Any `EventListeners` capable of responding to the event registered to objects above the `target` will process the event.
- **Bubbling** — The event starts propagation at its `target` and triggers any qualifying `EventListeners` registered to the `target`. The event then proceeds upward through the scripting object model, triggering any qualifying `EventListeners` registered to objects above the `target` in the scripting object model hierarchy.

The following table provides more detail on the properties of an `event` and the ways in which they relate to event propagation through the scripting object model.

| Property | Description |
|---|---|
| `Bubbles` | If true, the `event` propagates to scripting objects *above* the object initiating the `event`. |
| `Cancelable` | If true, the default behavior of the `event` on its `target` can be canceled. To do this, use the `PreventDefault` method. |
| `Captures` | If true, the `event` may be handled by `EventListeners` registered to scripting objects above the target object of the event during the capturing phase of event propagation. This means an `EventListener` on the application, for example, can respond to a document event before an `EventListener` is triggered. |
| `CurrentTarget` | The current scripting object processing the `event`. See `target` in this table. |
| `DefaultPrevented` | If true, the default behavior of the `event` on the current `target` (see `target` in this table) was prevented, thereby cancelling the action. |
| `EventPhase` | The current stage of the `event` propagation process. |
| `EventType` | The type of the `event`, as a string (for example, `"beforeNew"`). |
| `PropagationStopped` | If true, the `event` has stopped propagating beyond the `current target` (see `target` in this table). To stop event propagation, use the `StopPropagation` method. |
| `Target` | The object from which the `event` originates. For example, the `target` of a `beforeImport` event is a document; of a `beforeNew` event, the application. |
| `TimeStamp` | The time and date the `event` occurred. |

# Working with eventListeners

When you create an `EventListener`, you specify the event type (as a string) the event handler (as a file reference), and whether the `EventListener` can be triggered in the capturing phase of the event. The following script fragment shows how to add an `EventListener` for a specific event (for the complete script, see AddEventListener).

```
Set myInDesign = CreateObject("InDesign.Application.CS6"
Set myEventListener = myInDesign.EventListeners.Add("afterNew",
"c:\ICEventListeners\Message.vbs", false)
```

The script referred to in the above script contains the following code:

```
Rem "evt" is the event passed to this script by the event listener.
MsgBox ("This event is the " & evt.EventType & " event.")
```

To remove the `EventListener` created by the above script, run the following script (from the RemoveEventListener tutorial script):

```
Set myInDesign = CreateObject("InCopy.Application")
Set myFileSystemObject = CreateObject("Scripting.FileSystemObject")
Set myFile = myFileSystemObject.GetFile("c:\IDEventHandlers\message.vbs")
myResult = myInDesign.RemoveEventListener("afterNew", myFile, False)
```

When an `EventListener` responds an event, the event may still be processed by other `EventListeners` that might be monitoring the event (depending on the propagation of the event). For example, the `afterOpen` event can be observed by `EventListeners` associated with both the application and the document.

`EventListeners` do not persist beyond the current InCopy session. To make an `EventListener` available in every InCopy session, add the script to the startup scripts folder (for more on installing scripts, see Chapter 2, "Getting Started."). When you add an `EventListener` script to a document, it is not saved with the document or exported to INX.

**NOTE:** If you are having trouble with a script that defines an `EventListener`, you can either run a script that removes the `EventListener` or quit and restart InCopy.

An event can trigger multiple `EventListeners` as it propagates through the scripting object model. The following sample script demonstrates an event triggering `EventListeners` registered to different objects (for the full script, see MultipleEventListeners):

```
Set myInDesign = CreateObject("InCopy.Application")
Set myEventListener = myInDesign.EventListeners.Add("beforeImport",
"c:\EventInfo.vbs", True)
Set myDocument = myInDesign.Documents.Add
Set myEventListener = myDocument.EventListeners.Add("beforeImport",
"c:\EventInfo.vbs", False)
```

The `EventInfo.vbs` script referred to in the above script contains the following script code:

```
main evt
Function main(myEvent)
    myString = "Current Target: " & myEvent.CurrentTarget.Name
    MsgBox myString, vbOKOnly, "Event Details"
end function
```

When you run the preceding script and place a file, InCopy displays alerts showing, in sequence, the name of the document, then the name of the application.

The following sample script creates an `EventListener` for each supported event and displays information about the event in a simple dialog box. For the complete script, see EventListenersOn.

```
Rem EventListenersOn.vbs
Rem An InCopy CS6 JavaScript
Rem
Rem Installs event listeners for all supported events; displays a
Rem message when each event occurs.
Rem
Set myInCopy = CreateObject("InCopy.Application")
myEventNames = Array("beforeQuit", "afterQuit", "beforeNew", "afterNew", "beforeOpen",
"afterOpen", "beforeClose", "afterClose", "beforeSave", "afterSave", "beforeSaveAs",
"afterSaveAs", "beforeSaveACopy", "afterSaveACopy", "beforeRevert", "afterRevert",
"beforePrint", "afterPrint", "beforeExport", "afterExport", "beforeImport",
"afterImport", "beforePlace", "afterPlace")
For myCounter = 0 To UBound(myEventNames)
    myInCopy.AddEventListener myEventNames(myCounter), "c:\GetEventInfo.vbs", False
    If myCounter < UBound(myEventNames) Then
        myInCopy.EventListeners.Add myEventNames(myCounter), "c:\GetEventInfo.vbs",
False
    End If
Next
```

The following script is the one referred to by the preceding script. The file reference in the preceding script must match the location of this script on your disk. For the complete script, see `GetEventInfo.vbs`.

```
Rem GetEventInfo.vbs
Rem An InCopy CS6 VBScript
Rem
Rem Displays information about an Event object; called from an EventListener.
main evt
Function main(myEvent)
    myString = "Handling Event: " & myEvent.EventType
    myString = myString & vbCr & vbCr & "Target: " & myEvent.Target & " " &
    myEvent.Target.Name
    myString = myString & vbCr & "Current: " & myEvent.CurrentTarget & " " &
    myEvent.CurrentTarget.Name
    myString = myString & vbCr & vbCr & "Phase: " & myGetPhaseName(myEvent.EventPhase)
    myString = myString & vbCr & "Captures: " & myEvent.Captures
    myString = myString & vbCr & "Bubbles: " & myEvent.Bubbles
    myString = myString & vbCr & vbCr & "Cancelable: " & myEvent.Cancelable
    myString = myString & vbCr & "Stopped: " & myEvent.PropagationStopped
    myString = myString & vbCr & "Canceled: " & myEvent.DefaultPrevented
    myString = myString & vbCr & vbCr & "Time: " & myEvent.TimeStamp
    MsgBox myString, vbOKOnly, "Event Details"
end function
Rem Function returns a string corresponding to the event phase enumeration.
Function myGetPhaseName(myEventPhase)
    Select Case myEventPhase
        Case idEventPhases.idAtTarget
            myPhaseName = "At Target"
        Case idEventPhases.idBubblingPhase
            myPhaseName = "Bubbling"
        Case idEventPhases.idCapturingPhase
            myPhaseName = "Capturing"
        Case idEventPhases.idDone
            myPhaseName = "Done"
        Case idEventPhases.idNotDispatching
            myPhaseName = "Not Dispatching"
    end select
    myGetPhaseName = myPhaseName
End Function
```

The following sample script shows how to turn off all `EventListeners` for the application object. For the complete script, see EventListenersOff.

```
Rem EventListenersOff.vbs
Rem An InCopy CS6 JavaScript
Rem
Rem Removes all event listeners from the application.
Set myInCopy = CreateObject("InCopy.Application")
For myCounter = 1 To myInCopy.EventListeners.Count
    myInCopy.EventListeners.Item(1).Delete
Next
```

# A sample “afterNew” eventListener

The `afterNew` event provides a convenient place to add information to the document, like user name, document creation date, copyright information, and other job-tracking information. The following sample script shows how to add this sort of information to document metadata (also known as file info or XMP information). For the complete script listing, refer to the AfterNew tutorial script.

```
Set myInCopy = CreateObject("InCopy.Application")
Set myEventListener = myInCopy.EventListeners.Add("afterNew",
"c:\AfterNewHandler.vbs")
```

The following script is the one referred to by the preceding script. The file reference in the preceding script must match the location of this script on your disk. For the complete script, see `AfterNewHandler.vbs`.

```
Rem AfterNewHandler.vbs
Rem An InCopy CS6 VBScript
Rem
Rem Adds metadata to a new document.
myAddMetadata evt
Function myAddMetadata(myEvent)
    Set myInCopy = CreateObject("InCopy.Application")
    Set myDocument = myInCopy.Documents.Item(1)
    myInCopy.UserName = "Adobe"
    With myDocument.MetadataPreferences
        .Author = "Adobe Systems"
        .Description = "This is a sample document with XMP metadata." &
        vbCr & "Created: " + myEvent.TimeStamp
    End With
End Function
```

# 8 Notes

| Chapter Update Status | |
|---|---|
| CS6 | Unchanged |

With the InDesign and InCopy inline editorial-notes features, you can add comments and annotations as notes directly to text without affecting the flow of a story. Notes features are designed to be used in a workgroup environment. Notes can be color coded or turned on or off based on certain criteria.

Notes can be created using the Note tool in the toolbox, the Notes > New Note command, or the New Note icon on the Notes palette.

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script. We also assume you have some knowledge of working with notes in InCopy.

## Entering and importing a note

This section covers the process of getting a note into your InCopy document. Just as you can create a note and replace the text of the note using the InCopy user interface, you can create notes and insert text into a note using scripting.

### Adding a note to a story

To add note to a story, use the `Add` method. The following sample adds a note at the last insertion point. For the complete script, see InsertNote.

```
Set myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Rem We'll use the last insertion point in the story.
Set myInsertionPoint = myStory.insertionPoints.Item(-1)
Set myNote = myInsertionPoint.Notes.Add
myNote.Texts.Item(1).Contents = "This is a Note."
```

### Replacing text of a note

To replace the text of a note, use the `Contents` property, as shown in the following sample. For the complete script, see Replace.

```
myInCopy = CreateObject("InCopy.Application.CS6")
myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myNote = myStory.Notes.Item(1)
Replace text of note with "This is a replaced note."
myNote.Texts.Item(1).Contents = "This is a replaced note."
```

# Converting between notes and text

## Converting a note to text

To convert a note to text, use the `ConvertToText` method, as shown in the following sample. For the complete script, see ConvertToText.

```
Set myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
myNote = myStory.Notes.Item(1)
myNote.convertToText()
```

## Converting text to a note

To convert text to a note, use the `ConvertToNote` method, as shown in the following sample. For the complete script, see ConvertToNote.

```
Set myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
myStory.words.Item(1).convertToNote()
```

# Expanding and collapsing notes

## Collapsing a note

The following script fragment shows how to collapse a note. For the complete script, see CollapseNote.

```
Set myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myNote = myStory.Notes.Item(1)
myNote.Collapsed = True
```

## Expanding a note

The following script fragment shows how to expand a note. For the complete script, see ExpandNote.

```
myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myNote = myStory.Notes.Item(1)
myNote.Collapsed = False
```

# Removing a note

To remove a note, use the `Delete` method, as shown in the following sample. For the complete script, see RemoveNote.

```
Set myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myNote = myStory.Notes.Item(1)
myStory.Notes.Item(1).Delete
```

# Navigating among notes

## Going to the first note in a story

The following script fragment shows how to go to the first note in a story. For the complete script, see FirstNote.

```
Set myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myNote =  myStory.Notes.firstItem()
myNote.Texts.Item(1).Contents = "This is the first note."
```

## Going to the next note in a story

The following script fragment shows how to go to the next note in a story. For the complete script, see NextNote.

```
Set myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myNote =  myStory.Notes.nextItem(myStory.Notes.Item(1))
myNote.Texts.Item(1).Contents = "This is the next note."
```

## Going to the previous note in a story

The following script fragment shows how to go to the previous note in a story. For the complete script, see PreviousNote.

```
Set myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myNote =  myStory.Notes.previousItem(myStory.Notes.Item(2))
myNote.Texts.Item(1).Contents = "This is the prev note."
```

## Going to the last note in a story

The following script fragment shows how to go to the last note in a story. For the complete script, see LastNote.

```
Set myInCopy = CreateObject("InCopy.Application.CS6")
Set myDocument = myIncopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myNote =  myStory.Notes.lastItem()
myNote.Texts.Item(1).Contents = "This is the last note."
```

# 9 Tracking Changes

| Chapter Update Status | |
|---|---|
| CS6 | Unchanged |

Writers can track, show, hide, accept, and reject changes as a document moves through the writing and editing process. All changes are recorded and visualized to make it easier to review a document.

This chapter shows how to script the most common operations involving tracking changes.

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script. We also assume that you have some knowledge of working with text in InCopy and understand basic typesetting terms.

## Tracking Changes

This section shows how to navigate tracked changes, accept changes, and reject changes using scripting.

Whenever anyone adds, deletes, or moves text within an existing story, the change is marked in galley and story views.

### Navigating tracked changes

If the story contains a record of tracked changes, the user can navigate sequentially through tracked changes. The following scripts show how to navigate the tracked changes.

The followinng script uses the `nextItem` method to navigate to the change following the insertion point:

```
Set myDocument = myInCopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
//Story.trackChanges   If true, track changes is turned on.
If(myStory.TrackChanges=true ) Then
    Set myChange = myStory.Changes.Item(1)
    If(myStory.Changes.Count>1) Then
        Set myChange0 = myStory.Changes.NextItem(myChange)
    End If
End If
```

In the following script, we use the `previousItem` method to navigate to the change following the insertion point:

```
Set myDocument = myInCopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
If(myStory.TrackChanges=true ) Then
    Set myChange = myStory.Changes.LastItem()
    If(myStory.Changes.Count>1) Then
    Set myChange0 = myStory.Changes.PreviousItem(myChange)
    End If
End If
```

## Accepting and reject tracked changes

When changes are made to a story, by you or others, the change-tracking feature enables you to review all changes and decide whether to incorporate them into the story. You can accept and reject changes—added, deleted, or moved text—made by any user.

In the following script, the change is accepted:

```
Set myDocument = myInCopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myChange = myStory.Changes.Item(1)
myChange.Accept
```

In the following script, the change is rejected:

```
Set myDocument = myInCopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myChange = myStory.Changes.Item(1)
myChange.Reject
```

## Information about tracked changes

Change information includes include date and time. The following script shows the information of a tracked change:

```
Set myDocument = myInCopy.Documents.Item(1)
Set myStory = myDocument.Stories.Item(1)
Set myChange = myStory.Changes.Item(1)
With  myChange
    Rem idChangeTypes.idDeletedText  (Read Only) Deleted text.
    Rem idChangeTypes.idInsertedText  (Read Only) Insert text.
    Rem idChangeTypes.idMovedText  (Read Only) Moved text.
    myTypes = .ChangeType
    Rem Characters  A collection of Characters.
    Set myCharacters = .Characters
    Rem Character = myCharacters.Item(1);
    myDate = .Date
    Rem InsertionPoints A collection of insertion points.
    Rem  insertpoint = myInsertionPoints.Item(1);
    Set myInsertionPoints = .InsertionPoints
    Rem Lines  (Read Only) A collection of lines.
    Set myLines = .Lines
    Rem Paragraphs  (Read Only) A collection of paragraphs.
    Set myParagraphs =.Paragraphs
    Rem InsertionPoints A collection of insertion points.
```

```
    Rem  myInsertpoint = myInsertionPoints.Item(0);
    Set myStoryOffset = .StoryOffset
    Rem TtextColumns  (Read Only) A collection of text columns.
    Set myTextColumns = .TextColumns
    Rem TextStyleRanges  (Read Only) A collection of text style ranges.
    Set myTextStyleRanges = .TextStyleRanges
    Rem TextVariableInstances  (Read Only) A collection of text variable instances.
    Set myTextVariableInstances = .TextVariableInstances
    Rem Texts  (Read Only) A collection of text objects.
    Set myTexts = .Texts
    Rem The user who made the change. Note: Valid only when track changes is true.
    myUserName = .UserName
    Rem Words  A collection of words
    Set myWords = .Words
End With
```

# Preferences for tracking changes

Track-changes preferences are user settings for tracking changes. For example, you can define which changes are tracked (adding, deleting, or moving text). You can specify the appearance of each type of tracked change, and you can have changes identified with colored change bars in the margins. The following script shows how to set and get these preferences:

```
Set myTrackChangesPreference =  myInCopy.TrackChangesPreferences
With myTrackChangesPreference
Rem AddedBackgroundColorChoice As idChangeBackgroundColorChoices, The background color
option for added text.
Rem  idChangeBackgroundColorChoices, Background color options for changed text.
Rem idChangeBackgroundUsesChangePrefColor  The background color for changed text is the
same as the track changes preferences background color. For information, see background
color for added text, background color for deleted text, or background color for moved
text.
Rem  idChangeBackgroundUsesGalleyBackgroundColor  The background color for changed text
is the same as the galley background color.
Rem  idChangeBackgroundUsesUserColor  The background color for changed text is the same
as the color assigned to the current user.
    myAddedBackgroundColorChoice = .AddedBackgroundColorChoice
    .AddedBackgroundColorChoice =
idChangeBackgroundColorChoices.idChangeBackgroundUsesChangePrefColor
Rem idChangeTextColorChoices,Changed text color options.
Rem  Property AddedTextColorChoice As idChangeTextColorChoices, The color option for
added text.
Rem idChangeUsesChangePrefColor,The text color for changed text is the same as the text
color defined in track changes preferences. For information, see text color for added
text, text color for deleted text, or text color for moved text.
Rem idChangeUsesGalleyTextColor,The text color for changed text is the same as the
galley text color.
    myAddedTextColorChoice = .AddedTextColorChoice
    .AddedTextColorChoice = idChangeTextColorChoices.idChangeUsesChangePrefColor
Rem BackgroundColorForAddedText,The background color for added text, specified as an
InCopy UI color. Note: Valid only when added background color choice is change
background uses change pref color. Type: Array of 3 Doubles (0 - 255) or
idInCopyUIColors enumerator
    myBackgroundColorForAddedText = .BackgroundColorForAddedText
    .BackgroundColorForAddedText = idUIColors.idGray
Rem BackgroundColorForDeletedText, The background color for deleted text, specified as
an InCopy UI color. Note: Valid only when deleted background color choice is change
background uses change pref color
```

```
    myBackgroundColorForDeletedText = .BackgroundColorForDeletedText
    .BackgroundColorForDeletedText = idUIColors.idRed
Rem BackgroundColorForMovedText,The background color for moved text. Note: Valid only
when moved background color choice is change background uses change pref color
    myBackgroundColorForMovedText = .BackgroundColorForMovedText
    .BackgroundColorForMovedText = idUIColors.idPink
Rem ChangeBarColor, The change bar color, specified as an InCopy UI color.
    myChangeBarColor = .ChangeBarColor
    .ChangeBarColor = idUIColors.idCharcoal
Rem DeletedBackgroundColorChoice,The background color option for deleted text.
Rem idChangeBackgroundUsesChangePrefColor  The background color for changed text is the
same as the track changes preferences background color. For information, see background
color for added text, background color for deleted text, or background color for moved
text.
Rem  idChangeBackgroundUsesGalleyBackgroundColor  The background color for changed text
is the same as the galley background color.
Rem  idChangeBackgroundUsesUserColor  The background color for changed text is the same
as the color assigned to the current user.
    myDeletedBackgroundColorChoice = .DeletedBackgroundColorChoice
    .DeletedBackgroundColorChoice =
idChangeBackgroundColorChoices.idChangeBackgroundUsesUserColor
Rem DeletedTextColorChoice, The color option for deleted text.
Rem idChangeUsesChangePrefColor,The text color for changed text is the same as the text
color defined in track changes preferences. For information, see text color for added
text, text color for deleted text, or text color for moved text.
Rem idChangeUsesGalleyTextColor,The text color for changed text is the same as the
galley text color.
    myDeletedTextColorChoice = .DeletedTextColorChoice
    .DeletedTextColorChoice = idChangeTextColorChoices.idChangeUsesChangePrefColor
Rem LocationForChangeBar,The change bar location.
Rem  idChangebarLocations,Change bar location options.
Rem idLeftAlign, Change bars are in the left margin.
Rem idRightAlign, Change bars are in the right margin
    myLocationForChangeBar = .LocationForChangeBar
    .LocationForChangeBar = idChangebarLocations.idLeftAlign
Rem MarkingForAddedText, The marking that identifies added text.
Rem idChangeMarkings, Marking options for changed text.
Rem idOutline,   Outlines changed text.
Rem idNone, Does not mark changed text.
Rem idStrikethrough, Uses a strikethrough to mark changed text.
Rem idUnderlineSingle, Underlines changed text.
    myMarkingForAddedText = .MarkingForAddedText
    .MarkingForAddedText = idChangeMarkings.idStrikethrough
Rem MarkingForDeletedText, The marking that identifies deleted text.
Rem idChangeMarkings, Marking options for changed text.
Rem idOutline,   Outlines changed text.
Rem idNone, Does not mark changed text.
Rem idStrikethrough, Uses a strikethrough to mark changed text.
Rem idUnderlineSingle, Underlines changed text.
    myMarkingForDeletedText = .MarkingForDeletedText
    .MarkingForDeletedText = idChangeMarkings.idUnderlineSingle
Rem MarkingForMovedText, The marking that identifies moved text.
Rem idChangeMarkings, Marking options for changed text.
Rem idOutline,   Outlines changed text.
Rem idNone, Does not mark changed text.
Rem idStrikethrough, Uses a strikethrough to mark changed text.
Rem idUnderlineSingle, Underlines changed text.
    myMarkingForMovedText = .MarkingForMovedText
    .MarkingForMovedText =  idChangeMarkings.idOutline
Rem MovedBackgroundColorChoice,The background color option for moved text.
```

```
Rem idChangeBackgroundUsesChangePrefColor  The background color for changed text is the
same as the track changes preferences background color. For information, see background
color for added text, background color for deleted text, or background color for moved
text.
Rem  idChangeBackgroundUsesGalleyBackgroundColor  The background color for changed text
is the same as the galley background color.
Rem  idChangeBackgroundUsesUserColor  The background color for changed text is the same
as the color assigned to the current user.
    myMovedBackgroundColorChoice = .MovedBackgroundColorChoice
    .MovedBackgroundColorChoice =
idChangeBackgroundColorChoices.idChangeBackgroundUsesChangePrefColor
Rem MovedTextColorChoice, The color option for moved text.
Rem idChangeUsesChangePrefColor,The text color for changed text is the same as the text
color defined in track changes preferences. For information, see text color for added
text, text color for deleted text, or text color for moved text.
Rem idChangeUsesGalleyTextColor,The text color for changed text is the same as the
galley text color.
    myMovedTextColorChoice = .MovedTextColorChoice
    .MovedTextColorChoice = idChangeTextColorChoices.idChangeUsesChangePrefColor
Rem if true, displays added text.
    myShowAddedText = .ShowAddedText
    .ShowAddedText = true
Rem  If true, displays change bars.
     myShowChangeBars  = .ShowChangeBars
     .ShowChangeBars = true
Rem ShowDeletedText, If true, displays deleted text.
    myShowDeletedText = .ShowDeletedText
    .ShowDeletedText = true
Rem ShowMovedText,If true, displays moved text.
    myShowMovedText = .ShowMovedText
    .ShowMovedText = true
Rem SpellCheckDeletedText, If true, includes deleted text when using the Spell Check
command.
    mySpellCheckDeletedText = .SpellCheckDeletedText
    .SpellCheckDeletedText = true
Rem TextColorForAddedText, The color for added text, specified as an InCopy UI color.
Note: Valid only when added text color choice is change uses change pref color.
    myTextColorForAddedText =  .TextColorForAddedText
    .TextColorForAddedText =  idUIColors.idBlue
Rem  TextColorForDeletedText,The color for deleted text.
    myTextColorForDeletedText = .TextColorForDeletedText
    .TextColorForDeletedText = idUIColors.idYellow
Rem  TextColorForMovedText,The color for moved text.
    myTextColorForMovedText = .TextColorForMovedText
    .TextColorForMovedText =  idUIColors.idGreen
End With
```

# 10 Assignments

| Chapter Update Status | |
|---|---|
| CS6 | Unchanged |

An *assignment* is a container for text and graphics in an InDesign file that can be viewed and edited in InCopy. Typically, an assignment contains related text and graphics, such as body text, captions, and illustrations that make up a magazine article. Only InDesign can create assignments and assignment files.

This tutorial shows how to script the most common operations involving assignments.

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script.

## Assignment object

The section shows how to work with assignments and assignment files. Using scripting, you can open the assignment file and get assignment properties.

### Opening assignment files

The following script shows how to open an existing assignment file:

```
Rem Open an exist assignment file
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Open("c:\a.icma")
Set myAssignement = myDocument.Assignments.Item(1)
```

### Iterating through assignment properties

The following script fragment shows how to get assignment properties, such as the assignment name, user name, location of the assignment file, and export options for the assignment.

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Item(1)
Set myAssignement = myDocument.Assignments.Item(1)
myuserName = myAssignement.UserName
myFilePath = myAssignement.FilePath
myDocPath = myAssignement.DocumentPath
myFramecolor = myAssignement.FrameColor
myincludeLinksWhenPackage = myAssignement.IncludeLinksWhenPackage
Rem Export options for assignment files.
Rem AssignmentExportOptions.ASSIGNED_SPREADS Exports only spreads with assigned frames
Rem AssignmentExportOptions.EMPTY_FRAMES  Exports frames but does not export content
Rem AssignmentExportOptions.EVERYTHING   Exports the entire document.
myExportOptions = myAssignement.ExportOptions
```

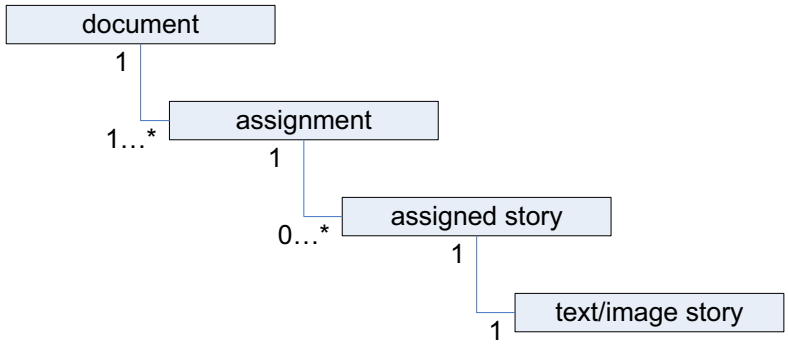
## Assignment packages

Assignment packages (`.incp` files created by InCopy) are compressed folders that contain assignment files. An assignment can be packaged using the `createPackage` method. The following sample script uses this technique to create a package file:

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Item(1)
Set myAssignement = myDocument.Assignments.Item(1)
If  myAssignement.Packaged = False Then
Rem idPackageType.idForwardPackage Creates an assignment package for export.
Rem idPackageType.idReturnPackage Create a package to place in the main document.
myAssignement.CreatePackage("c:\b.icap",idPackageType.idForwardPackage)
End If
```

# An assignment story

The following diagram shows InCopy's assignment object model. An assignment document contains one or more assignments; an assignment contains zero, one, or more assigned stories. Each assigned story references a text story or image story.




This section covers the process of getting assigned stories and assignment story properties.

## Assigned-story object

The following script shows how to get an assigned story from an assignment object:

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Item(1)
Set myAssignement = myDocument.Assignments.Item(1)
Set myAssignmentStory = myAssignement.AssignedStories.item(1)
```

## Iterating through the assigned-story properties

In InCopy, assigned-story objects have properties. The following script shows how to get all properties of an assigned-story object:

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Item(1)
Set myAssignement = myDocument.Assignments.Item(1)
Set myAssignmentStory = myAssignement.AssignedStories.item(1)
myName = myAssignmentStory.Name
myFilePath = myAssignmentStory.FilePath
Set myStoryReference = myAssignmentStory.StoryReference
```

# 11 XML

| Chapter Update Status | |
|---|---|
| CS6 | Unchanged |

Extensible Markup Language, or XML, is a text-based mark-up system created and managed by the World Wide Web Consortium (www.w3.org). Like Hypertext Markup Language (HTML), XML uses angle brackets to indicate markup tags (for example, `<article>` or `<para>`). While HTML has a predefined set of tags, XML allows you to describe content more precisely by creating custom tags.

Because of its flexibility, XML increasingly is used as a format for storing data. InCopy includes a complete set of features for importing XML data into page layouts, and these features can be controlled using scripting.

We assume that you have already read Chapter 2, "Getting Started" and know how to create, install, and run a script. We also assume that you have some knowledge of XML, DTDs, and XSLT.

## Overview

Because XML is entirely concerned with content and explicitly *not* concerned with formatting, making XML work in a page-layout context is challenging. InCopy's approach to XML is quite complete and flexible, but it has a few limitations:

- Once XML elements are imported into an InCopy document, they become InCopy elements that correspond to the XML structure. *The InCopy representations of the XML elements are not the same thing as the XML elements themselves.*
- Each XML element can appear only once in a layout. If you want to duplicate the information of the XML element in the layout, you must duplicate the XML element itself.
- The order in which XML elements appear in a layout depends largely on the order in which they appear in the XML structure.
- Any text that appears in a story associated with an XML element becomes part of that element's data.

## The best approach to scripting XML in InCopy

You might want to do most of the work on an XML file outside InCopy, before importing the file into an InCopy layout. Working with XML outside InCopy, you can use a wide variety of excellent tools, like XML editors and parsers.

When you need to rearrange or duplicate elements in a large XML data structure, the best approach is to transform the XML using XSLT. You can do this as you import the XML file.

# Scripting XML Elements

This section shows how to set XML preferences and XML import preferences, import XML, create XML elements, and add XML attributes. The scripts in this section demonstrate techniques for working with the XML content itself; for scripts that apply formatting to XML elements, see "Adding XML elements to a story" on page 103.

## Setting XML preferences

You can control the appearance of the InCopy structure panel using the XML view-preferences object, as shown in the following script fragment (from the XMLViewPreferences tutorial script):

```
Set myXMLViewPreferences = myDocument.XMLViewPreferences
myXMLViewPreferences.ShowAttributes = True
myXMLViewPreferences.ShowStructure = True
myXMLViewPreferences.ShowTaggedFrames = True
myXMLViewPreferences.ShowTagMarkers = True
myXMLViewPreferences.ShowTextSnippets = True
```

You also can specify XML tagging-preset preferences (the default tag names and user-interface colors for tables and stories) using the XML-preferences object, as shown in the following script fragment (from the XMLPreferences tutorial script):

```
Set myXMLPreferences = myDocument.XMLPreferences
myXMLPreferences.DefaultCellTagColor = idUIColors.idBlue
myXMLPreferences.DefaultCellTagName = "cell"
myXMLPreferences.DefaultImageTagColor = idUIColors.idBrickRed
myXMLPreferences.DefaultImageTagName = "image"
myXMLPreferences.DefaultStoryTagColor = idUIColors.idCharcoal
myXMLPreferences.DefaultStoryTagName = "text"
myXMLPreferences.DefaultTableTagColor = idUIColors.idCuteTeal
myXMLPreferences.DefaultTableTagName = "table"
```

## Setting XML import preferences

Before importing an XML file, you can set XML-import preferences that can apply an XSLT transform, govern the way white space in the XML file is handled, or create repeating text elements. You do this using the XML import-preferences object, as shown in the following script fragment (from the XMLImportPreferences tutorial script):

```
Set myXMLImportPreferences = myDocument.XMLImportPreferences
myXMLImportPreferences.AllowTransform = False
myXMLImportPreferences.CreateLinkToXML = False
myXMLImportPreferences.IgnoreUnmatchedIncoming = True
myXMLImportPreferences.IgnoreWhitespace = True
myXMLImportPreferences.ImportCALSTables = True
myXMLImportPreferences.ImportStyle = idXMLImportStyles.idMergeImport
myXMLImportPreferences.ImportTextIntoTables = False
myXMLImportPreferences.ImportToSelected = False
myXMLImportPreferences.RemoveUnmatchedExisting = False
myXMLImportPreferences.RepeatTextElements = True
Rem The following properties are only used when the
Rem AllowTransform property is set to True.
Rem myXMLImportPreferences.TransformFilename = "c:\myTransform.xsl"
Rem If you have defined parameters in your XSL file, then you can pass
Rem parameters to the file during the XML import process. For each parameter,
Rem enter an array containing two strings. The first string is the name of the
Rem parameter, the second is the value of the parameter.Rem
myXMLImportPreferences.TransformParameters = Array(Array("format", "1"))
```

## Importing XML

Once you set the XML-import preferences the way you want them, you can import an XML file, as shown in the following script fragment (from the ImportXML tutorial script):

```
myDocument.ImportXML("c:\completeDocument.xml")
```

When you need to import the contents of an XML file into a specific XML element, use the importXML method of the XML element, rather than the corresponding method of the document. See the following script fragment (from the ImportXMLIntoElement tutorial script):

```
Set myXMLTag = myDocument.XMLTags.Add("xml_element")
Set myRootXMLElement = myDocument.XMLElements.Item(1)
set myMXLElement = myRootElement.XMLElements.Add(myXMLTag)
myRootXMLElement.ImportXML "c:\completeDocument.xml"
```

You also can set the `ImportToSelected` property of the `XMLImportPreferences` object to true, then select the XML element, and then import the XML file, as shown in the following script fragment (from the ImportXMLIntoSelectedXMLElement tutorial script):

```
Set myDocument = myInDesign.Documents.Add
myDocument.ImportXML "c:\test.xml"
Set myRootXMLElement = myDocument.XMLElements.Item(1)
Set myLastXMLElement = myRootXMLElement.XMLElements.Item(-1)
Rem Select the XML element
myDocument.Select myLastXMLElement, idSelectionOptions.idReplaceWith
myDocument.XMLImportPreferences.ImportToSelected = True
myDocument.ImportXML "c:\test.xml"
```

## Creating an XML tag

XML tags are the names of XML elements that you want to create in a document. When you import XML, the element names in the XML file are added to the list of XML tags in the document. You also can create XML tags directly, as shown in the following script fragment (from the MakeXMLTags tutorial script):

```
Rem You can create an XML tag without specifying a color for the tag.
Set myXMLTagA = myDocument.XMLTags.Add("XML_tag_A")
Rem You can define the highlight color of the XML tag using the UIColors enumeration...
Set myXMLTagB = myDocument.XMLTags.Add("XML_tag_B", UIColors.Gray)
Rem ...or you can provide an RGB array to set the color of the tag.
Set myXMLTagC = myDocument.XMLTags.Add("XML_tag_C", Array(0, 92, 128))
```

## Loading XML tags

You can import XML tags from an XML file without importing the XML contents of the file. You might want to do this to work out a tag-to-style or style-to-tag mapping before importing the XML data, as shown in the following script fragment (from the LoadXMLTags tutorial script):

```
myDocument.LoadTags("c:\test.xml")
```

## Saving XML tags

Just as you can load XML tags from a file, you can save XML tags to a file, as shown in the following script. When you do this, only the tags themselves are saved in the XML file; document data is not included. As you would expect, this process is much faster than exporting XML, and the resulting file is much smaller. The following sample script shows how to save XML tags (for the complete script, see SaveXMLTags):

```
myDocument.SaveXMLTags("c:\xml_tags.xml", "Tag set created October 5, 2006")
```

## Creating an XML element

Ordinarily, you create XML elements by importing an XML file, but you also can create an XML element using InCopy scripting, as shown in the following script fragment (from the CreateXMLElement tutorial script):

```
Set myXMLTag = myDocument.XMLTags.Add("myXMLTag")
Set myRootElement = myDocument.XMLElements.Item(1)
Set myXMLElement = myRootElement.XMLElements.Add(myXMLTag)
myXMLElement.Contents = "This is an XML element containing text."
```

## Moving an XML element

You can move XML elements within the XML structure using the `move` method, as shown in the following script fragment (from the MoveXMLElement tutorial script):

```
Set myDocument = myInCopy.Documents.Add
Set myXMLTag = myDocument.XMLTags.Add("myXMLTag")
Set myRootElement = myDocument.XMLElements.Item(1)
Set myXMLElementA = myRootElement.XMLElements.Add(myXMLTag)
myXMLElementA.Contents = "This is XML element A."
Set myXMLElementB = myRootElement.XMLElements.Add(myXMLTag)
myXMLElementB.Contents = "This is XML element B."
myXMLElementA.Move idLocationOptions.idAfter, myXMLElementB
```

## Deleting an XML element

Deleting an XML element removes it from both the layout and the XML structure, as shown in the following script fragment (from the DeleteXMLElement tutorial script):

```
myRootXMLElement.XMLElements.Item(1).delete
```

## Duplicating an XML element

When you duplicate an XML element, the new XML element appears immediately after the original XML element in the XML structure, as shown in the following script fragment (from the DuplicateXMLElement tutorial script):

```
Set myDocument = myInCopy.Documents.Add
Set myXMLTag = myDocument.XMLTags.Add("myXMLTag")
Set myRootElement = myDocument.XMLElements.Item(1)
Set myXMLElementA = myRootElement.XMLElements.Add(myXMLTag)
myXMLElementA.Contents = "This is XML element A."
Set myXMLElementB = myRootElement.XMLElements.Add(myXMLTag)
myXMLElementB.Contents = "This is XML element B."
myXMLElementA.Duplicate
```

## Removing items from the XML structure

To break the association between a text object and an XML element, use the `untag` method, as shown in the following script. The objects are not deleted, but they are no longer tied to an XML element (which is deleted). Any content of the deleted XML element becomes associated with the parent XML element. If the XML element is the root XML element, any layout objects (text or page items) associated with the XML element remain in the document. (For the complete script, see UntagElement.)

```
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(-2)
myXMLElement.Untag
```

## Creating an XML comment

XML comments are used to make notes in XML data structures. You can add an XML comment using something like the following script fragment (from the MakeXMLComment tutorial script):

```
Set myRootElement = myDocument.XMLElements.Item(1)
Set myXMLElement = myRootElement.XMLElements.Add(myXMLTag)
Set myXMLComment = myXMLElement.XMLComments.Add("This is an XML comment.")
```

## Creating an XML processing instruction

A processing instruction (PI) is an XML element that contains directions for the application reading the XML document. XML processing instructions are ignored by InCopy but can be inserted in an InCopy XML structure for export to other applications. An XML document can contain multiple processing instructions.

An XML processing instruction has two parts, target and value. The following is an example of an XML processing instruction:

```
<?xml-stylesheet type="text/css" href="generic.css"?>
```

The following script fragment shows how to add an XML processing instruction (for the complete script, see MakeProcessingInstruction):

```
Set myRootXMLElement = myDocument.XMLElements.Item(1)
myRootXMLElement.XMLInstructions.Add "xml-stylesheet type=""text/css""",
"href=""generic.css"""
```

## Working with XML attributes

XML attributes are "metadata" that can be associated with an XML element. To add an attribute to an element, use something like the following script fragment. An XML element can have any number of XML attributes, but each attribute name must be unique within the element (that is, you cannot have two attributes named "id").

The following script fragment shows how to add an XML attribute to an XML element (for the complete script, see MakeXMLAttribute):

```
Set myDocument = myInCopy.Documents.Add
Set myXMLTag = myDocument.XMLTags.Add("myXMLElement")
Set myRootElement = myDocument.XMLElements.Item(1)
Set myXMLElement = myRootElement.XMLElements.Add(myXMLTag)
Set myXMLAttribure = myXMLElement.XMLAttributes.Add("example_attribute", "This is an
XML attribute.")
```

In addition to creating attributes directly using scripting, you can convert XML elements to attributes. When you do this, the text contents of the XML element become the value of an XML attribute added to the parent of the XML element. Because the name of the XML element becomes the name of the attribute, this method can fail when an attribute with that name already exists in the parent of the XML element. If the XML element contains page items, those page items are deleted from the layout.

When you convert an XML attribute to an XML element, you can specify the location where the new XML element is added. The new XML element can be added to the beginning or end of the parent of the XML attribute. By default, the new element is added at the beginning of the parent element.

You also can specify an XML mark-up tag for the new XML element. If you omit this parameter, the new XML element is created with the same XML tag as the XML element containing the XML attribute.

The following script shows how to convert an XML element to an XML attribute (for the complete script, see the ConvertElementToAttribute tutorial script):

```
Set myXMLTag = myDocument.XMLTags.Add("myXMLElement")
Set myRootXMLElement = myDocument.XMLElements.Item(1)
Set myXMLElement = myRootXMLElement.XMLElements.Add(myXMLTag)
Set myTargetXMLElement = myXMLElement.XMLElements.Add(myXMLTag)
myTargetXMLElement.Contents = "This is content in an XML element."
myTargetXMLElement.ConvertToAttribute
```

You also can convert an XML attribute to an XML element, as shown in the following script fragment (from the ConvertAttributeToElement tutorial script):

```
Set myXMLTag = myDocument.XMLTags.Add("myXMLElement")
Set myRootXMLElement = myDocument.XMLElements.Item(1)
Set myXMLElement = myRootXMLElement.XMLElements.Add(myXMLTag)
Set myXMLattribute = myXMLElement.XMLAttributes.Add("xml_attribute", "This is content
in an XML attribute.")
myXMLAttribute.ConvertToElement idXMLElementLocation.idElementEnd, myXMLTag
```

## Working with XML stories

When you import XML elements that were not associated with a layout element (a story or page item), they are stored in an XML story. You can work with text in unplaced XML elements just as you would work with the text in a text frame. The following script fragment shows how this works (for the complete script, see XMLStory):

```
Set myXMLStory = myDocument.XmlStories.Item(1)
Rem Though the text has not yet been placed in the layout, all text
Rem properties are available.
myXMLStory.Texts.Item(1).PointSize = 72
Rem Place the Root XML element in the default story so that
Rem you can see the result in the Structure panel.
myDocument.Stories.Item(1).PlaceXML myRootXMLElement
```

## Exporting XML

To export XML from an InCopy document, export either the entire XML structure in the document or one XML element (including any child XML elements it contains). The following script fragment shows how to do this (for the complete script, see ExportXML):

```
myDocument.Export "XML", "c:\test.xml"
```

# Adding XML elements to a story

Previously, we covered the process of getting XML data into InCopy documents and working with the XML structure in a document. In this section, we discuss techniques for getting XML information into a story and applying formatting to it.

## Associating XML elements with text

To associate text with an existing XML element, use the `PlaceXML` method. This replaces the content of the page item with the content of the XML element, as shown in the following script fragment (from the PlaceXML tutorial script):

```
myDocument.Stories.Item(1).PlaceXML(myXMLElements.Item(0)
```

To associate an existing text object with an existing XML element, use the `markup` method. This merges the content of the text object with the content of the XML element (if any). The following script fragment shows how to use the `markup` method (for the complete script, see Markup):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Add
Set myRootXMLElement = myDocument.XMLElements.Item(1)
Set myStory = myDocument.Stories.Item(1)
Rem Place the Root XML element in the default story.
myStory.PlaceXML myRootXMLElement
myString = "This is the first paragraph in the story." & vbCr
myString = myString & "This is the second paragraph in the story." & vbCr
myString = myString & "This is the third paragraph the story." & vbCr
myString = myString & "This is the fourth paragraph in the story." & vbCr
myStory.Contents = myString
Set myXMLTag = myDocument.XMLTags.Add("myXMLElement")
Set myXMLElement = myRootXMLElement.XMLElements.Add(myXMLTag)
Rem Mark up one of the paragraphs with another XML element.
myDocument.Stories.Item(1).Paragraphs.Item(3).Markup myXMLElement
```

### Inserting text in and around XML text elements

When you place XML data into an InCopy story, you often need to add white space (for example, return and tab characters) and static text (labels like “name” or “address”) to the text of your XML elements. The

following sample script shows how to add text in and around XML elements (for the complete script, see InsertTextAsContent):

```
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(1)
Rem By inserting the return character after the XML element, the character
Rem becomes part of the content of the parent XML element,
Rem not of the element itself.
myXMLElement.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(2)
myXMLElement.InsertTextAsContent "Static text: ", idXMLElementPosition.idBeforeElement
myXMLElement.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Rem To add text inside the element, set the location option to beginning or end.
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(3)
myXMLElement.InsertTextAsContent "Text at the start of the element: ",
idXMLElementPosition.idElementStart
myXMLElement.InsertTextAsContent " Text at the end of the element.",
idXMLElementPosition.idElementEnd
myXMLElement.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Rem Add static text outside the element.
Set myXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(4)
myXMLElement.InsertTextAsContent "Text before the element: ",
idXMLElementPosition.idBeforeElement
myXMLElement.InsertTextAsContent " Text after the element.",
idXMLElementPosition.idAfterElement
Rem To insert text inside the text of an element, work with the text objects contained
by the element.
myXMLElement.Words.Item(2).InsertionPoints.Item(1).Contents = "(the third word of) "
```

## Mapping tags to styles

One of the quickest ways to apply formatting to XML text elements is to use `XMLImportMaps`, also known as tag-to-style-mappings. When you do this, you can associate a specific XML tag with a paragraph or character style. When you use the`MapTagsToStyles` method of the document, InCopy applies the style to the text, as shown in the following script fragment (from the MapTagsToStyles tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Item(1)
Rem Create a tag to style mapping.
myDocument.XMLImportMaps.Add myDocument.XMLTags.Item("heading_1"),
myDocument.ParagraphStyles.Item("heading 1")
myDocument.XMLImportMaps.Add myDocument.XMLTags.Item("heading_2"),
myDocument.ParagraphStyles.Item("heading 2")
myDocument.XMLImportMaps.Add myDocument.XMLTags.Item("para_1"),
myDocument.ParagraphStyles.Item("para 1")
myDocument.XMLImportMaps.Add myDocument.XMLTags.Item("body_text"),
myDocument.ParagraphStyles.Item("body text")
Rem Apply the XML tag to style mapping.
myDocument.MapXMLTagsToStyles
```

## Mapping styles to tags

When you have formatted text that is not associated with any XML elements, and you want to move that text into an XML structure, use style-to-tag mapping, which associates paragraph and character styles with XML tags. To do this, use `XMLExportMaps` objects to create the links between XML tags and styles, then use the `MapStylesToTags` method to create the corresponding XML elements, as shown in the following script fragment (from the MapStylesToTags tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Item(1)
Rem Create a tag to style mapping.
myDocument.XMLExportMaps.Add myDocument.ParagraphStyles.Item("heading 1"),
myDocument.XMLTags.Item("heading_1")
myDocument.XMLExportMaps.Add myDocument.ParagraphStyles.Item("heading 2"),
myDocument.XMLTags.Item("heading_2")
myDocument.XMLExportMaps.Add myDocument.ParagraphStyles.Item("para 1"),
myDocument.XMLTags.Item("para_1")
myDocument.XMLExportMaps.Add myDocument.ParagraphStyles.Item("body text"),
myDocument.XMLTags.Item("body_text")
Rem Apply the tag to style mapping.
myDocument.MapStylesToXMLTags
```

Another approach is simply to have your script create a new XML tag for each paragraph or character style in the document, and then apply the style to tag mapping, as shown in the following script fragment (from the MapAllStylesToTags tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Item(1)
Rem Create tags that match the style names in the document,
Rem creating an XMLExportMap for each tag/style pair.
For myCounter = 1 To myDocument.ParagraphStyles.Count
    Set myParagraphStyle = myDocument.ParagraphStyles.Item(myCounter)
    myParagraphStyleName = myParagraphStyle.Name
    myXMLTagName = Replace(myParagraphStyleName, " ", "_")
    myXMLTagName = Replace(myXMLTagName, "[", "")
    myXMLTagName = Replace(myXMLTagName, "]", "")
    Set myXMLTag = myDocument.XMLTags.Add(myXMLTagName)
    myDocument.XMLExportMaps.Add myParagraphStyle, myXMLTag
Next
Rem Apply the tag to style mapping.
myDocument.MapStylesToXMLTags
```

## Applying styles to XML elements

In addition to using tag-to-style and style-to-tag mappings or applying styles to the text and page items associated with XML elements, you also can apply styles to XML elements directly. The following script fragment shows how to use the methods `ApplyParagraphStyle` and `ApplyCharacterStyle`. (For the complete script, see ApplyStylesToXMLElements.)

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Item(1)
Rem Add XML elements.
Set myRootXMLElement = myDocument.XMLElements.Item(1)
Set myXMLElementA =
myRootXMLElement.XMLElements.Add(myDocument.XMLTags.Item("heading_1"))
myXMLElementA.Contents = "Heading 1"
myXMLElementA.ApplyParagraphStyle myDocument.ParagraphStyles.Item("heading 1"), True
myXMLElementA.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Set myXMLElementB =
myRootXMLElement.XMLElements.Add(myDocument.XMLTags.Item("para_1"))
myXMLElementB.Contents = "This is the first paragraph in the article."
myXMLElementB.ApplyParagraphStyle myDocument.ParagraphStyles.Item("para 1"), True
myXMLElementB.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Set myXMLElementC =
myRootXMLElement.XMLElements.Add(myDocument.XMLTags.Item("body_text"))
myXMLElementC.Contents = "This is the second paragraph in the article."
```

```
myXMLElementC.ApplyParagraphStyle myDocument.ParagraphStyles.Item("body text"), True
myXMLElementC.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Set myXMLElementD =
myRootXMLElement.XMLElements.Add(myDocument.XMLTags.Item("heading_2"))
myXMLElementD.Contents = "Heading 2"
myXMLElementD.ApplyParagraphStyle myDocument.ParagraphStyles.Item("heading 2"), True
myXMLElementD.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Set myXMLElementE =
myRootXMLElement.XMLElements.Add(myDocument.XMLTags.Item("para_1"))
myXMLElementE.Contents = "This is the first paragraph following the subhead."
myXMLElementE.ApplyParagraphStyle myDocument.ParagraphStyles.Item("para 1"), True
myXMLElementE.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Set myXMLElementF =
myRootXMLElement.XMLElements.Add(myDocument.XMLTags.Item("body_text"))
myXMLElementF.Contents = "This is the second paragraph following the subhead."
myXMLElementF.ApplyParagraphStyle myDocument.ParagraphStyles.Item("body text"), True
myXMLElementF.InsertTextAsContent vbCr, idXMLElementPosition.idAfterElement
Set myXMLElementG =
myXMLElementF.XMLElements.Add(myDocument.XMLTags.Item("body_text"))
myXMLElementG.Contents = "Note:"
Set myXMLElementG = myXMLElementG.Move(idLocationOptions.idAtBeginning, myXMLElementF)
myXMLElementG.InsertTextAsContent " ", idXMLElementPosition.idAfterElement
myXMLElementG.ApplyCharacterStyle myDocument.CharacterStyles.Item("Emphasis"), True
Set myStory = myDocument.Stories.Item(1)
Rem Associate the root XML element with the story.
myRootXMLElement.PlaceXML myStory
```

## Working with XML tables

InCopy automatically imports XML data into table cells when the data is marked up using HTML standard table tags. If you cannot or prefer not to use the default table mark-up, InCopy can convert XML elements to a table using the `ConvertElementToTable` method.

To use this method, the XML elements to be converted to a table must conform to a specific structure. Each row of the table must correspond to a specific XML element, and that element must contain a series of XML elements corresponding to the cells in the row. The following script fragment shows how to use this method (for the complete script, see ConvertXMLElementToTable). The XML element used to denote the table row is consumed by this process.

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Add
Rem Create a series of XML tags.
Set myRowTag = myDocument.XMLTags.Add("row")
Set myCellTag = myDocument.XMLTags.Add("cell")
Set myTableTag = myDocument.XMLTags.Add("table")
Rem Add XML elements.
Set myRootXMLElement = myDocument.XMLElements.Item(1)
With myRootXMLElement
    Set myTableXMLElement = .XMLElements.Add(myTableTag)
    With myTableXMLElement
        For myRowCounter = 1 To 6
            With .XMLElements.Add(myRowTag)
                myString = "Row " & CStr(myRowCounter)
                For myCellCounter = 1 To 4
                    With .XMLElements.Add(myCellTag)
                        .Contents = myString & ":Cell " & CStr(myCellCounter)
                    End With
                Next
            End With
        Next
    End With
End With
Set myTable = myTableXMLElement.ConvertElementToTable(myRowTag, myCellTag)
Set myStory = myDocument.Stories.Item(1)
myStory.PlaceXML myDocument.XMLElements.Item(1)
```

Once you are working with a table containing XML elements, you can apply table styles and cell styles to the XML elements directly, rather than having to apply the styles to the tables or cells associated with the XML elements. To do this, use the `applyTableStyle` and `applyCellStyle` methods, as shown in the following script fragment (from the ApplyTableStyle tutorial script):

```
Set myInCopy = CreateObject("InCopy.Application")
Set myDocument = myInCopy.Documents.Add
Rem Create a series of XML tags.
Set myRowTag = myDocument.XMLTags.Add("row")
Set myCellTag = myDocument.XMLTags.Add("cell")
Set myTableTag = myDocument.XMLTags.Add("table")
Rem Create a table style and a cell style.
Set myTableStyle = myDocument.TableStyles.Add
myTableStyle.StartRowFillColor = myDocument.Colors.Item("Black")
myTableStyle.StartRowFillTint = 25
myTableStyle.EndRowFillColor = myDocument.Colors.Item("Black")
myTableStyle.EndRowFillTint = 10
Set myCellStyle = myDocument.CellStyles.Add
myCellStyle.FillColor = myDocument.Colors.Item("Black")
myCellStyle.FillTint = 45
Rem Add XML elements.
Set myRootXMLElement = myDocument.XMLElements.Item(1)
With myRootXMLElement
    Set myTableXMLElement = .XMLElements.Add(myTableTag)
    With myTableXMLElement
        For myRowCounter = 1 To 6
            With .XMLElements.Add(myRowTag)
                myString = "Row " + CStr(myRowCounter)
                For myCellCounter = 1 To 4
                    With .XMLElements.Add(myCellTag)
                        .Contents = myString & ":Cell " + CStr(myCellCounter)
                    End With
                Next
```

```
            End With
        Next
    End With
End With
Set myTable = myTableXMLElement.ConvertElementToTable(myRowTag, myCellTag)
Set myTableXMLElement = myDocument.XMLElements.Item(1).XMLElements.Item(1)
myTableXMLElement.ApplyTableStyle myTableStyle
myTableXMLElement.XMLElements.Item(1).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(6).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(11).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(16).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(17).ApplyCellStyle myCellStyle
myTableXMLElement.XMLElements.Item(22).ApplyCellStyle myCellStyle
myDocument.Stories.Item(1).PlaceXML myDocument.XMLElements.Item(1)
myTable.AlternatingFills = idAlternatingFillsTypes.idAlternatingRows
```